An Object Server for the Process Handbook

by

Umar Farooq

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirement of the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1997

Author_____
Department of Electrical Engineering and Computer Science
May 16, 1997

Certified by_____
Professor Thomas W. Malone
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An Object Server for the Process Handbook

by

Umar Farooq

Submitted to the
Department of Electrical Engineering and Computer Science

May 16, 1997

In Partial Fulfillment of the Requirement of the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Over the last few years, the Process Handbook software had grown rapidly in functionality (and size). The database for the current version was not implemented with a clean modular interface providing an appropriate abstraction. This thesis explains the design and implementation of an OLE (Object Linking and Embedding) server for the Process Handbook. The new design introduces significant modularity and abstraction into the Process Handbook design. More specifically, the new design: (1) enables easier extensions to the Process Handbook, which include extensions to the Handbook's algorithms and development of new clients for the Handbook, (2) allows easier maintenance of the integrity of the underlying database, (3) permits full Internet access to the Handbook, (4) enables easier implementation of security features, (5) makes the Handbook forward compatible, (6) allows easier interaction with other software packages, (9) provides a small and transparent API (Application Programming Interface) to the clients, (10) enables full attribute and dependency inheritance, and (11) facilitates the import and export of complex processes. All in all, the new design contributes towards making the Process Handbook a useful, extensible, and robust tool for analyzing processes.

Thesis Supervisor: Professor Thomas W. Malone
Title: Patrick J. McGovern Professor of Information Systems

Acknowledgments

I would like to thank many people for their support and guidance throughout my graduate work. First, I would like to thank my thesis supervisor, Professor Thomas Malone, for his guidance and support, both academic and personal, throughout the last one year. My most sincere thanks to John Quimby for his immeasurable help with my work, particularly for his support during the Object API design meetings. Thanks to Mark Klein, Chris Dellarocas, Avi Bernstein, Calvin Yuen, and Voung Nguyen for their valuable contributions to the project. I would also like to thank the people at the Center for Coordination Science, who made it a very fun and interesting place to work and learn. My deepest thanks to my family, particularly my mother (whose guidance and love are the primary reasons for my academic success), for their support, confidence, and love. Thanks to my friends, particularly, Junaid, Samad, Hisham, and Zia, for their encouragement throughout my stay at M.I.T. Most of all, I would like to thank Allah for all the opportunities and successes that He granted me over the years.

I dedicate all my work (past, present, and future) to the memories of my *Amma* (Begum Riaz Razzaq) and my *Abba* (Chaudhry Abdul Razzaq), the two most important people in my life. Though they are no longer with me, their memories will always live with me.
May their souls rest in Heaven.

Table of Contents

Chapter 1

Introduction

The Process Handbook project at the MIT Center for Coordination Science involves collecting examples of how different organizations perform similar processes, and organizing these processes in an on-line "Process Handbook" (Malone, et al., 1997). The Process Handbook is intended to help people: (1) redesign existing organizational processes, (2) invent new organizational processes, (3) learn about organizations, and (4) automatically generate software to support organizational processes. The methodology used in the Process Handbook is semantically very rich. It allows the users to represent and analyze complex processes. These processes can also be viewed in many of the common process representation formats, for example, IDEF0, data flow, etc.

The Process Handbook methodology uses concepts from coordination science and computer science (such as inheritance). All processes in the Process Handbook are categorized in a specialization hierarchy (with very generic processes at the top and increasingly specialized processes at the lower levels). The users can use the specialization hierarchy to understand the deep structure of processes and to invent more specialized versions of existing processes. The more complex processes in the handbook have other simpler processes in their decomposition. This allows the users to view any process at various levels of detail. A process in the Handbook can also have any number of attributes which define the key characteristics of the process. The specializations of a process inherit both its attributes and its decomposition. Finally, the Process Handbook methodology uses the notion of coordination science that coordination can be viewed as managing dependencies between different processes.

The implementation of the Handbook that is currently being used by the users (this thesis discusses a new implementation that is still be developed and is not available to the users) was developed over quite a few years by many software developers. Over these years, the software has grown tremendously in size, complexity, and functionality. Moreover, some

of the developers working on the Process Handbook did not use the existing code modules in routines that they added to the code base. This lead to the "reinvention of the wheel" quite a few times. As a result, the current version of the Process Handbook, in its interface to the process repository (database), lacks modularity and abstraction, both of which are very desirable in a complex system. Although, the current implementation (developed for the MS-Windows platform) of the Process Handbook provides a lot of functionality to the users, the lack of modularity and abstraction has made it very cumbersome to add more functionality to the Handbook. There were two efforts in the past to introduce modularity and abstraction by using a 'middleware' API, but they were not too successful. Previous versions of Visual Basic (the language used for the development) did not support object-oriented programming so the past efforts could only simulate a business logic API.

The work described in this thesis explains the design and implementation of an object server for the Process Handbook that exposes OLE (Object Linking and Embedding) objects. This server has been designed in a very modular manner which allows easy extensions to the Handbook functionality. Software developers can manipulate the OLE objects, through a simple, small, and transparent API, to develop clients for the viewing and editing of the information stored in the Process Handbook. The API abstracts the implementation of algorithms in the server as well as the details of the database schema.

Many scientists at the Center for Coordination Science, namely Prof. Thomas Malone, Prof. Chris Dellarocas, John Quimby, Mark Klein, and Abraham Bernstein, took a very active part in the design of this API. Whenever I refer to 'us' or 'we' in this document, the reader should understand that I mean all of the above mentioned people along with myself.

Background concerning coordination theory and the semantics of the Process Handbook is provided in chapter 2. The goals of the new system are stated in chapter 3. A brief overview of the Object API is presented in chapter 4. Chapter 4 can be used as a guide

by prospective users of the Object API. The implementation of the API is discussed in much greater detail in chapter 5. An evaluation of the new design, with respect to the goals for the new system, is presented in chapter 6. Future work that could enhance the functionality of the Process Handbook even further is outlined in chapter 7. The conclusions that have been drawn after the work on this project are presented in chapter 8.

Chapter 2

Background

2.1 Coordination Theory

To understand the methodology of the Process Handbook, it is extremely important to understand the underlying concepts of coordination science. This is because coordination science forms the backbone of the Process Handbook. Without coordination theory, the Process Handbook would be much less effective as a tool for understanding and inventing new processes.

Coordination theory is an emerging research area that deals with the interdisciplinary study of coordination. The research in this area uses concepts from such varied fields as computer science, organization theory, operations research, economics, linguistics, and psychology (Malone and Crowston, 1994). Intuitively, coordination could be thought as being able to work together. Another simple, yet more precise, definition of coordination is that: coordination is managing dependencies between activities (I will be using the words 'process' and 'activity' interchangeably in this thesis).

The Process Handbook uses the more precise view of coordination. Processes in the Process Handbook can be decomposed into sub-processes and the dependencies between these sub-processes (this is discussed in much greater detail in section 2.2). The dependencies between these sub-processes can be managed by managing activities (processes that manage dependencies). Being able to view and alter the managing activities for the dependencies gives the users of the Process Handbook the power to invent new processes, understand existing processes in more detail, and improve the existing processes. The inventors of the Process Handbook, and the scientists working in the area of coordination science, subscribe to the view that the dependencies and their managing activities are much more important than the sub-processes in a decomposition.

The Process Handbook can be used to invent new processes by using a four steps. First, the user needs to specify the high level goals (parent processes). Second, these processes are decomposed into sub-processes. Third, the dependencies between the various sub-processes are specified. Finally, and most importantly, the managing activities for these dependencies are specified. The last step specifies how the 'coordination' between the sub-processes takes place.

2.2 The Process Handbook

The Process Handbook has grown significantly in functionality over the last few years. For example, it now contains some objects (attribute-type, Navigational Node etc.) that are primarily for efficiency in the users' interaction with the Process Handbook. A more detailed analysis of these objects is presented in chapter 5. This section describes the activities, dependencies, ports, connectors, attributes, and bundles (along with the specializations and decomposition of these objects) which form the backbone of the semantic model of the Process Handbook.

2.2.1 Decomposition of Activities

Activities can be decomposed into sub-activities. The decomposition of activities allows the users view the activity at many levels of detail. While one user might be interested only in the high level activity, another user might want to see how the activity can be broken down into the very basic activities. Theoretically, any activity can be decomposed infinitely, but the users who add content to the Process Handbook only decompose the activities (that they add) to a level that they find desirable and useful.

Figure 1 illustrates a very simple decomposition of the activity *Sell Product*. The decomposition of this activity consists of activities such as *Identify Potential Customers* and *Inform Potential Customers*. However, it is obvious that Figure 1 cannot be used as a complete specification for the *Sell Product* process. The specification cannot be complete without dependencies For instance, we need to identify potential customers

before we can inform them and we need to obtain the order before we can receive the payment.

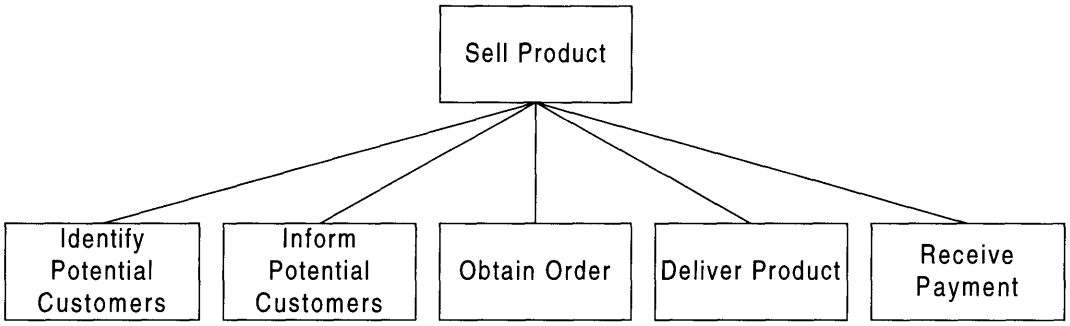


Figure 1. The decomposition of *Sell Product.*

2.2.2 Dependencies, Ports, and Connectors

Dependencies form an integral part of the Process Handbook methodology. Dependencies represent the interaction between different activities. Figure 2 illustrates the three basic kinds of dependencies: *flow*, *sharing*, and *fit*. Flow dependencies arise when a resource produced by one activity is consumed by another activity. Sharing dependencies arise when a single resource is consumed by multiple activities. Fit dependencies occur when multiple activities produce a single resource (Malone, et al., 1997).

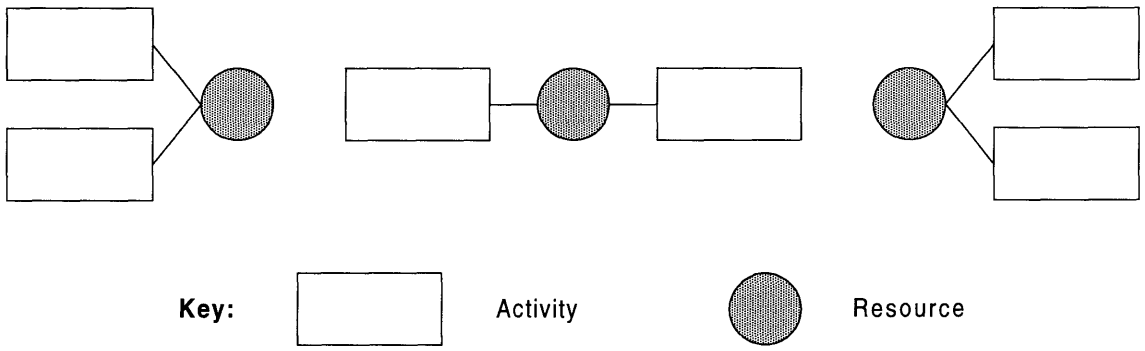


Figure 2. Three basic types of dependencies among activities (adapted from Zlotkin, 1995)

Figure 3 shows a simple flow dependency. The figure shows the decomposition of the process *Make Coffee* into two sub-activities: fetching the coffee beans and brewing coffee. The dependency is needed to specify three things: the beans have to be fetched before the coffee can be brewed (prerequisite constraint), the beans have to be transported to the place where the coffee will be brewed (accessibility constraint), and the brewing mechanism should be able to use the coffee beans that are fetched (usability constraint). This is a very simple dependency which only has one producer and one consumer activity. Dependencies can become quite complicated when they interact with multiple producers and consumer activities.

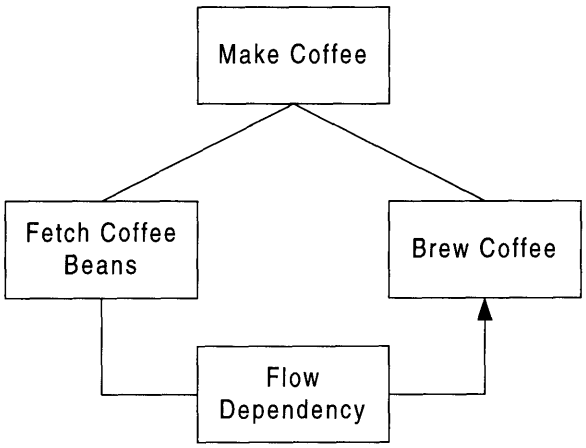


Figure 3. A simple flow dependency.

In the Process Handbook, the activities and dependencies interact with each other through their ports. Ports can be thought of as inlets and outlets for an activity or dependency. An activity or a dependency can have any number of ports. In the *Make Coffee* example, the activity *Fetch Coffee Beans* would have a port that would be the outlet for the coffee beans that have been fetched. Similarly, the activity *Brew Coffee* would have a port through which the coffee beans are consumed. The *Flow Dependency* would also have ports that are used in the mediation of the coffee beans that flow from *Fetch Coffee Beans* to *Brew Coffee*. Figure 4 shows the full semantic structure of the decomposition in Figure 3. As figure 4 shows, ports of activities and dependencies are also part of their

decomposition. A parent activity is related to the sub-activities, dependencies, and ports in its decomposition by decomposition relations (decomposition relations are explained in more detail in chapters 4 and 5).

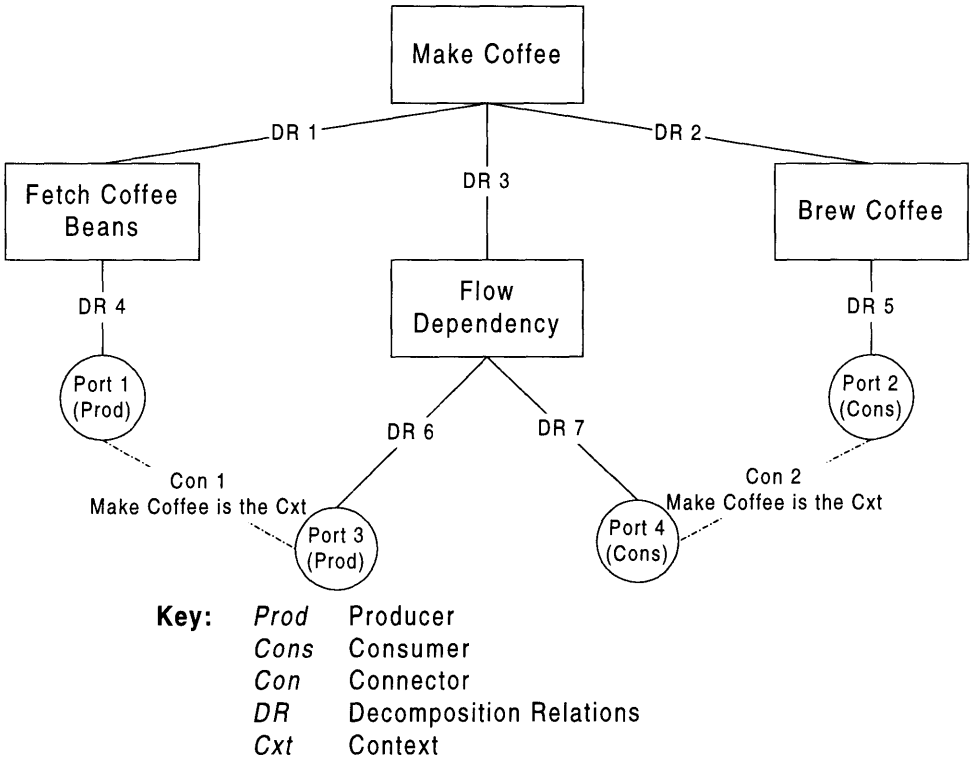


Figure 4. The semantic structure of the decomposition in Figure 3

Ports of different activities and dependencies are connected to each other by connectors. There can be two kinds of connectors. The horizontal connectors connect the ports of siblings in a decomposition i.e., the activities and dependencies that exist at the same level in a decomposition. The vertical connectors connect the ports of an activity and the ports of the sub-activities in its decomposition. Figure 4 only illustrates horizontal connectors because the *Flow Dependency*, *Fetch Coffee Beans*, and *Brew Coffee* are siblings in the decomposition of *Make Coffee*. Vertical connectors play an important role in dependencies that are not between siblings in a decomposition (the example, later in

this section, illustrating the setting of managing activities for dependencies will also illustrate vertical connectors).

The Process Handbook uses the notion that *coordination* can be defined as *managing dependencies among activities*. The users of the Process Handbook have the ability to set managing activities for the dependencies represented in a decomposition. The power of analyzing processes is also enhanced greatly by the ability to replace the existing managing activities with other managing activities from the repository of managing activities in the Process Handbook. When a user initially sets the managing activity for a dependency, that dependency is replaced by simpler dependencies. Figures 5a and 5b illustrate the changes in the semantic representation when a managing activity is set for a dependency.

Figure 5a (overleaf) shows the two level decomposition of an activity *A*. *Depe 1* is the only dependency in this decomposition. It has two producers (*D* and *E*) and a single consumer (*F*). *Con 4*, *Con 5*, and *Con6* are vertical connectors that connect the ports *Port 1* with *Port 4*, *Port 2* with *Port 5*, and *Port 3* with *Port 6* respectively. The vertical connectors serve two purposes. First, they simplify the implementation of algorithms required for rendering dependencies (see Elley, 1996 for more detail). Second, and more importantly, they relate the ports on an activity with the ports on the sub-activities. For example, in Figure 5a, *Con 4* and *Con 5* explain which ports of the sub-activities of *B* correspond to the ports on *B*. Connectors exist in the context of an activity. The horizontal connectors have the decomposition parent of the dependency as their context, while the vertical connectors have the parent activity as their context.

Figure 5b (overleaf) illustrates the results of setting the managing activity *M* for the dependency *Depe 1*. All the connectors, decomposition relations, and activities in the decompositions of *B* and *C* are unchanged by this operation. I have not included them in the diagram to avoid undue cluttering. The dependencies, ports, decomposition relations, and connectors that are created as a result of this operation are drawn with thicker lines.

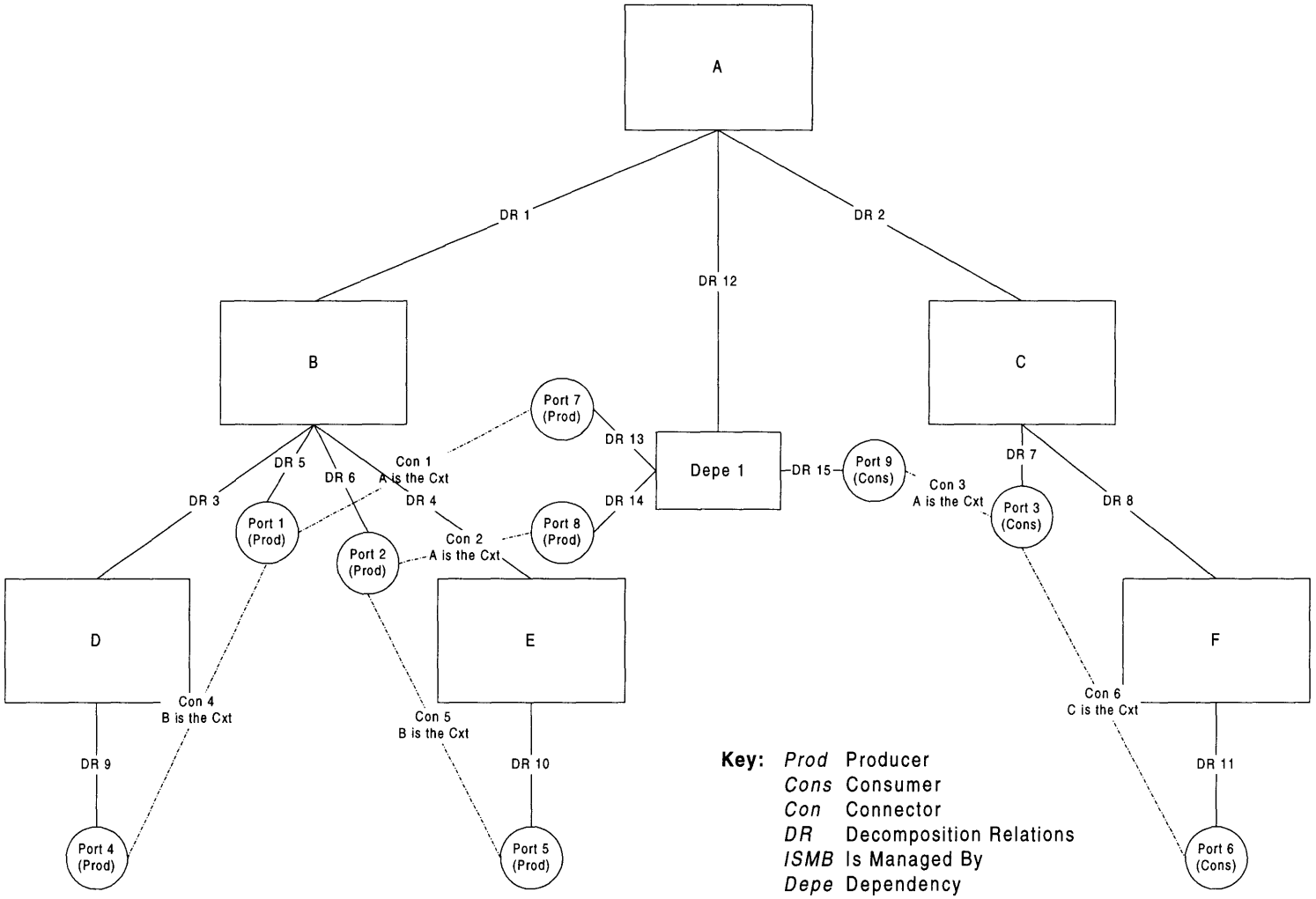


Figure 5a. The Semantics of a dependency (Depe 1) with multiple producers and a single consumer

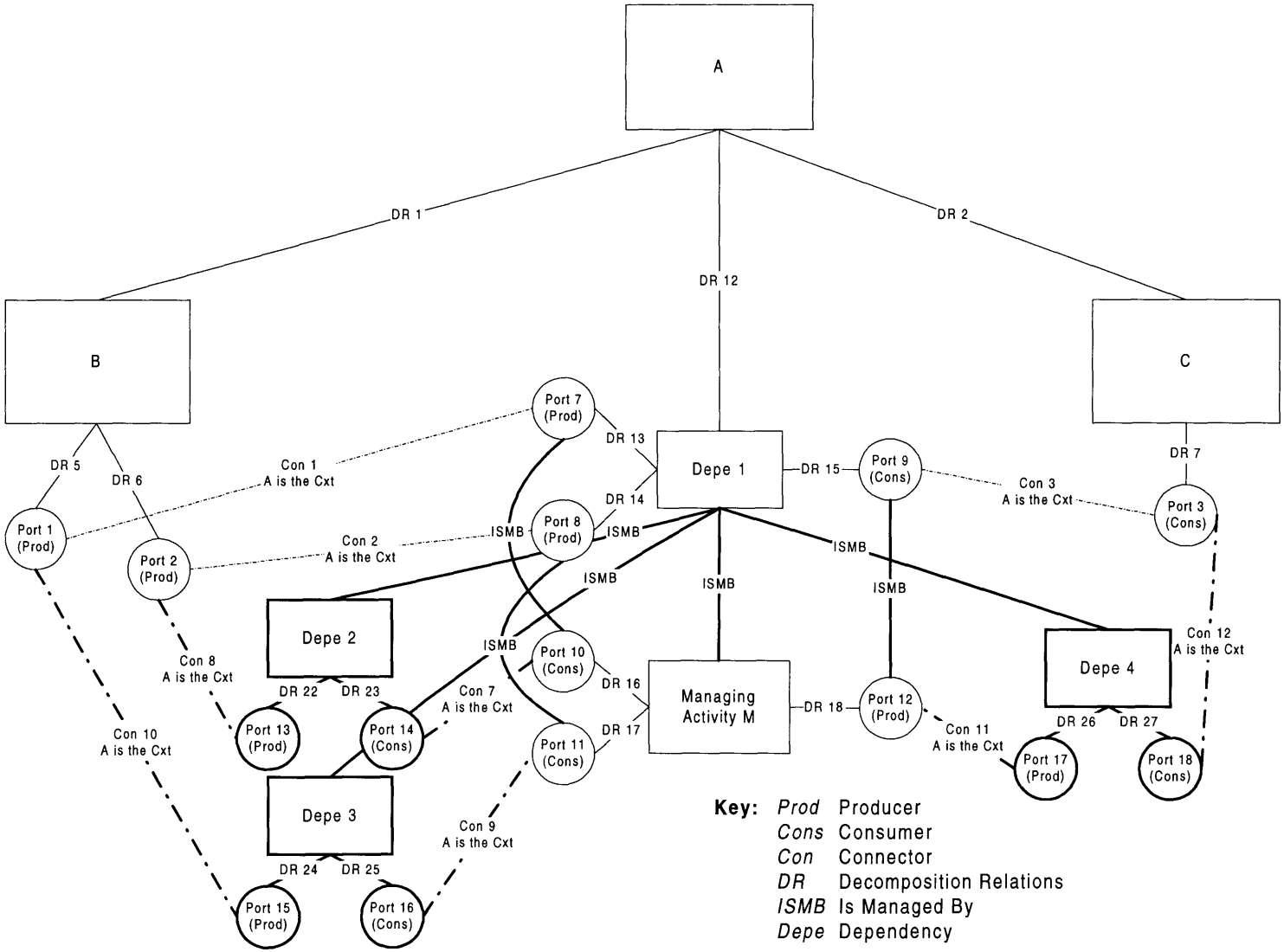


Figure 5b. The changes to the decomposition of A after a managing activity is set for the dependency Depe 1.

The user has to choose a managing activity from the existing library (the user could also create a specialization of an existing managing activity) of managing activities in the Process Handbook and explicitly match the ports of the managing activity with the ports of the dependency (this is not required if the dependency only has a single producer and a single consumer because the system does the matching automatically). The original dependency is broken into simpler dependencies as a result of setting a managing activity because managing a dependency simplifies it. In general, if the user sets a managing activity for a dependency with x producers and y consumers, $x + y$ new simple (with one producer and one consumer) dependencies are created.

Figure 5a and 5b show the semantic representation of a dependency with two producers and one consumer. I decided not to use the most general case (with both multiple producers and multiple consumers) to avoid cluttering the diagram. However, the model presented by figures 5a and 5b can be easily extended to the most general case.

2.2.3 Specializations of Activities

The activities in the Process Handbook span two dimensions: the dimension of generality and the dimension of detail. Figure 6 illustrates these two dimensions.

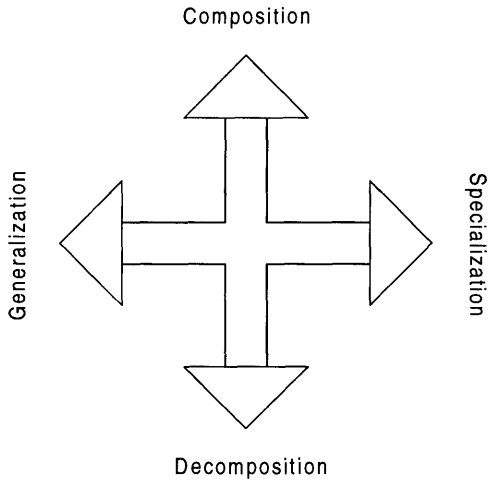


Figure 6. The Dimensions for Expressing Activities

Figure 6 can be seen a compass in which the North-South dimension represents the composition and decomposition of an activity. For example, the activity *Make Coffee* can have the activity *Fetch Coffee Beans* in its decomposition; at the same time *Make Coffee* itself can be in the decomposition of the activity *Prepare Breakfast.* Most of the people are used of thinking about activities only in terms of the Composition-Decomposition (or the North-South) dimension.

The Process Handbook methodology also allows the characterization of activities in the Generalization-Specialization dimension. This, combined with inheritance (inheritance is discussed in greater detail in section 2.2.6), makes the Process Handbook a very powerful tool. An activity in the Process Handbook can have many specializations and generalizations. *Make Coffee* can have the activity *Make Iced Coffee* as a specialization and the activity *Make* as a generalization. All the activities in the Process Handbook are part of a specialization hierarchy which has very generic processes at the top level and increasingly specialized processes at lower levels.

Figure 7 illustrates how the two dimensions work together in enhancing the power of the Process Handbook.

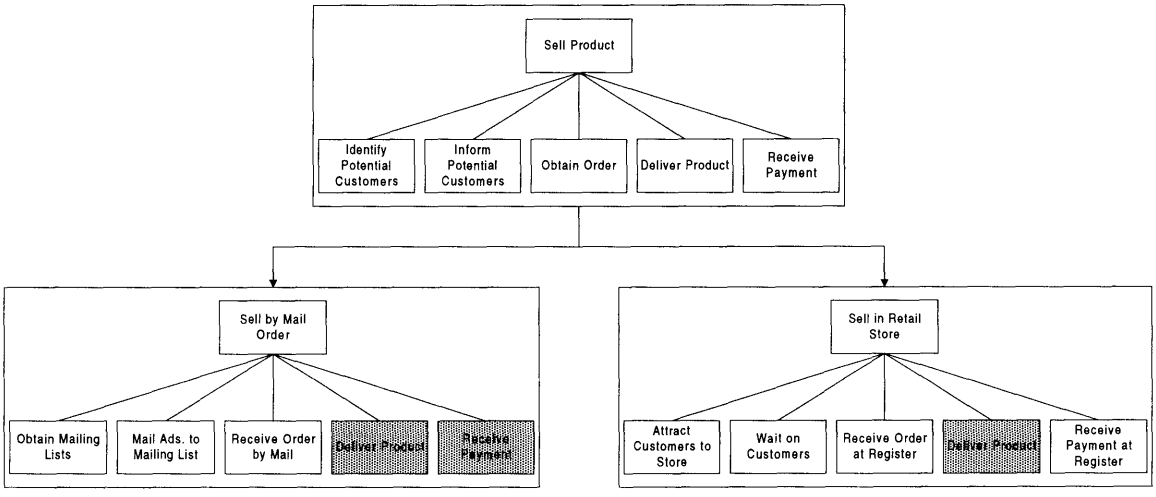


Figure 7. The representation of three different sales processes. *Sell by Mail Order* and *Sell in Retail Store* are the specializations of the generic process *Sell Product.* The shaded sub-activities are inherited without change. (adapted from Malone, et al., 1997).

The two specializations of the *Sell Product* activity inherit its decomposition. Some of the sub-activities of *Sell by Mail Order* and *Sell in Retail Store* have to be further specialized to fully express these activities. For instance, the activity *Identify Potential Customers* is replaced by a more specialized activity *Obtain Mailing Lists* in the decomposition of *Sell by Mail Order*. Other sub-activities might not need to be changed. The shaded sub-activities in figure 7 are inherited without any change. Besides the advantages of combining the specializations and decomposition, this approach allows conciseness of representations and generative representations (Malone, et al., 1997).

2.2.4 Attributes

All the entities (activities, ports, dependencies, navigational nodes, attribute-types, bundles, and resources) in the Process Handbook have attributes. Some of these attributes distinguish the entity from the other entities in the Process Handbook. *Name*, *ID*, *PIFID* (Process Interchange Format ID), etc. are some of such attributes. The other attributes of the entities might not distinguish them, rather these attributes help in explaining the entity. *Actors involved in an activity*, *description*, *skills required to perform the activity*, etc. are some such attributes.

Some of the attributes are system-generated (the attributes that uniquely identify the entity are among these attributes). For instance, the attribute Name is generated by the system for each newly created entity. The user also has the ability to add any attributes that she might find necessary. For example, the user might want to include the names of the people who are interested in a particular activity as an attribute of that activity. The implementation of the attributes in the server is discussed in more detail in chapters 4 and 5.

2.2.5 Bundles

The specializations of a process are arranged in bundles. Each bundle contains a set of alternative processes. Bundles significantly enhance the power of the Process Handbook. Figure 8 shows an example of grouping specializations in bundles. The specializations of the activity *Sell Something* are arranged in two bundles (shaded boxes). It should be

clarified that the bundles are not a part of the specialization hierarchy. The bundles are actually attributes of the activity *Sell Something* and the activities in the bundles are attributes of the those bundles (this is explained in more detail in chapters 4 and 5).

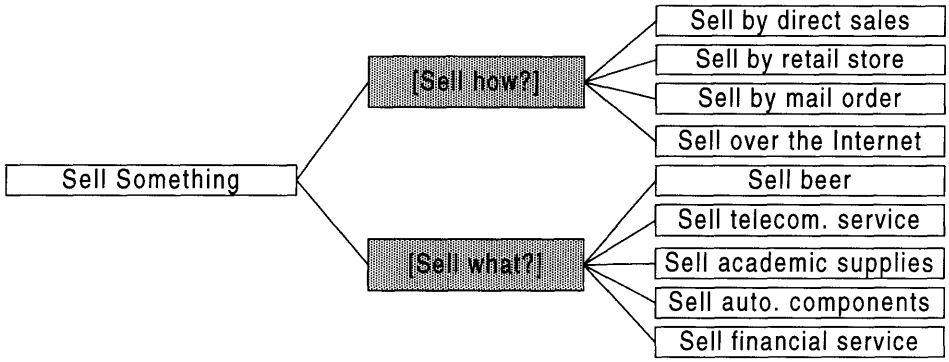


Figure 8. Specializations of the activity *Sell Something* arranged in the bundles *Sell how?* and *Sell what?* (Adapted from Malone, et al, 1997)

Bundles are primarily used in two ways. First, they are used for comparing the alternatives. These comparisons are facilitated by trade-off matrices that compare the important characteristics of the alternatives (these trade-off matrices are generated by using the information stored as the attributes of the bundles). For example, an interested user could match the cost of selling by direct sales and the cost of selling by retail store (if cost of selling is one of the characteristics compared in the trade-off matrix for the *Sell how?* bundle). Second, the bundles are used for restricting certain kinds of inheritance. Alternatives in a bundle can only inherit alternatives from other bundles. For example, someone selling beer should be presented with the alternatives for direct mail, retail store, and selling over the Internet, but it does not make much sense for her to be presented with alternatives of selling auto. components or selling financial service.

2.2.6 Inheritance

The inheritance in the Process Handbook considerably increases its power and value to users. The specializations of an entity inherit its attributes and decomposition. The Process Handbook borrows the concept of inheritance from object-oriented systems. Whereas, traditional object-oriented systems have a hierarchy of objects where the

specialized objects inherit the functionality of the more general objects, the Process Handbook has a hierarchy of both verbs (activities) and objects (the resources in the Process Handbook specialization hierarchy are actually objects) that behave in a similar fashion. It is quite important to understand the representational semantics of the inheritance of attributes and decomposition in the Process Handbook.

(i) The Inheritance of Attributes

The specializations of an entity can inherit its attributes in four different ways. First, some of the attributes might not be inherited at all by the specializations. Second, some of the attributes might be inherited with a pre-specified default value. For example, if user adds an attribute to specify the color for an activity and wanted all the newly created specializations to be gray, she could use this kind of inheritance. If she sets the color of an activity to be red and creates a specialization of that activity, the newly created specialization would still be gray. Third, some of the attributes might be inherited without a value. For example, if there were an attribute that evaluated a particular activity after its performance, it might be inherited in this fashion. Finally, and most commonly, some attributes might be inherited along with their value.

(ii)The Inheritance of the Decomposition of an Entity

Inheritance of the decomposition of entities is quite complicated because it involves: inheritance of activities and dependencies, inheritance of ports, and the inheritance of connectors. Figures 9a, 9b, and 9c illustrate the inheritance of a decomposition. Figure 9a (overleaf) shows the decomposition of an activity *A*. The decomposition is two level deep and there is a single producer-single consumer dependency between *D* and *E*.

Figure 9b (overleaf) shows what happens when we create *A1*, a specialization of *A*. The decomposition of *A1* still has the same *B*, *C*, *Depe 1*, *D*, and *E* as *A's* decomposition (new decomposition relations are created between *A1* and *B* and between *A1* and *C*).

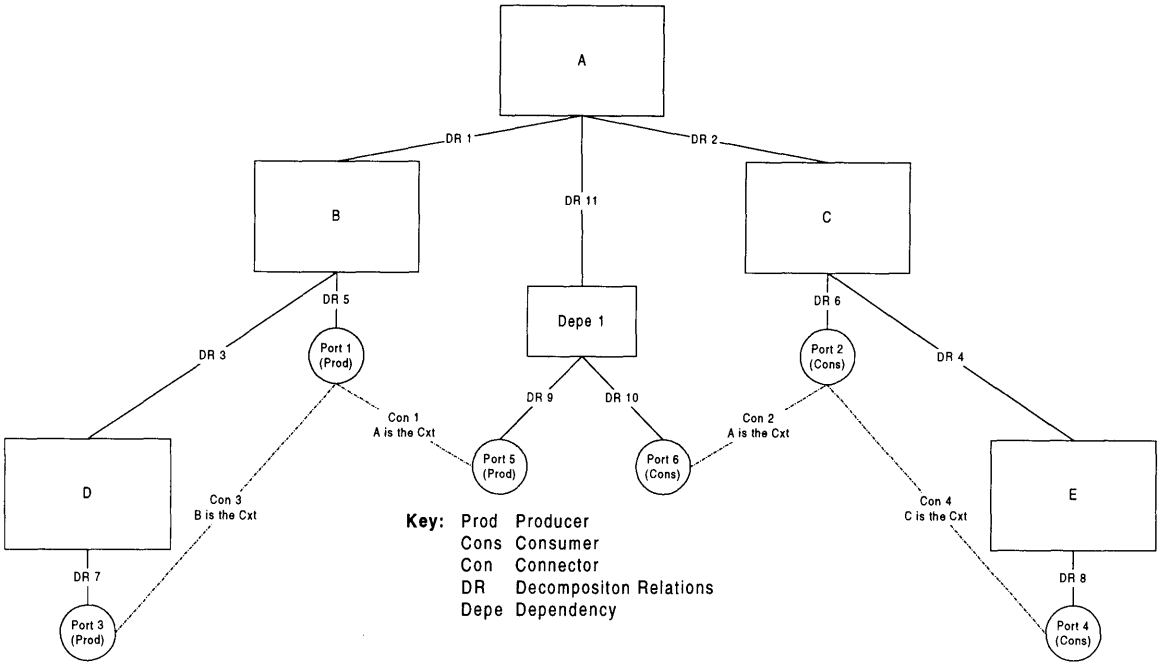


Figure 9a. The two level decomposition of the activity *A*.

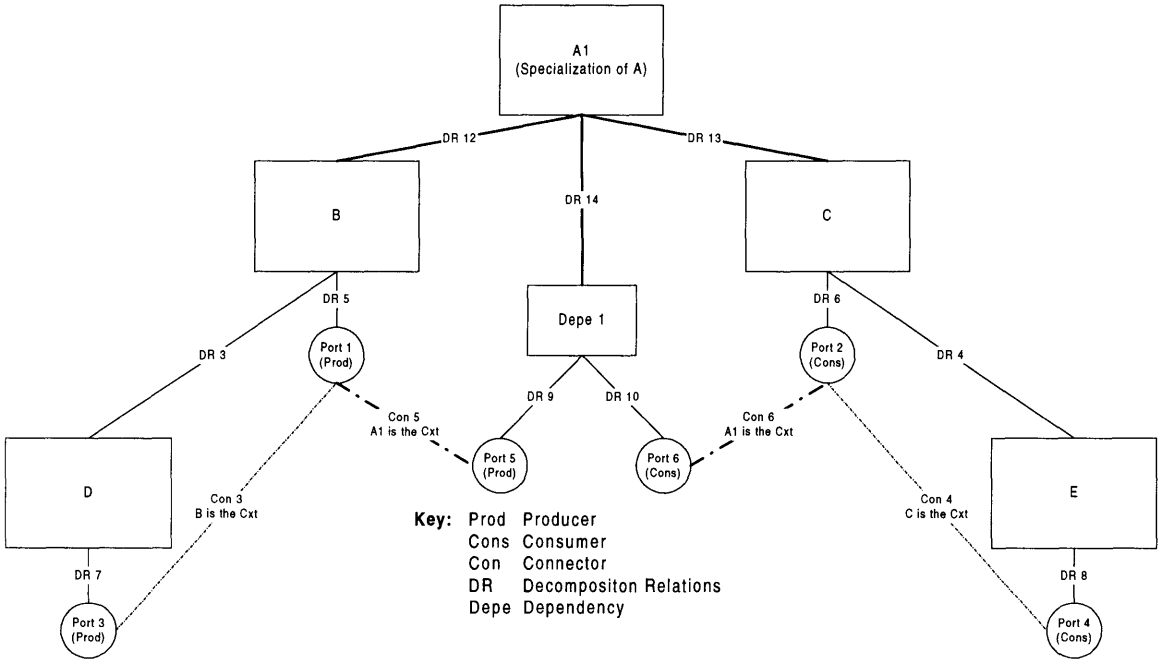


Figure 9b. *A1* is a specialization of *A*. It inherits the decomposition of *A*. Heavy lines represent the newly created decomposition relations.

It is important that *A1* inherit *B*, *C*, and *Depe 1* by reference because if new changes were made to *B*, *C*, *Depe 1*, *D*, or *E* in the context of *A*'s decomposition, those changed would be automatically be reflected in the decomposition of *A1*. However the connectors *Con 1*

and *Con 2*, which exist in the context of *A*, are replaced by connectors *Con 5* and *Con 6* respectively. The connectors have to be replaced because they are very strongly tied with their context. Whereas, sub-activities are not dependent on their parent activity, the connectors only have any meaning in context of the parent of the decomposition.

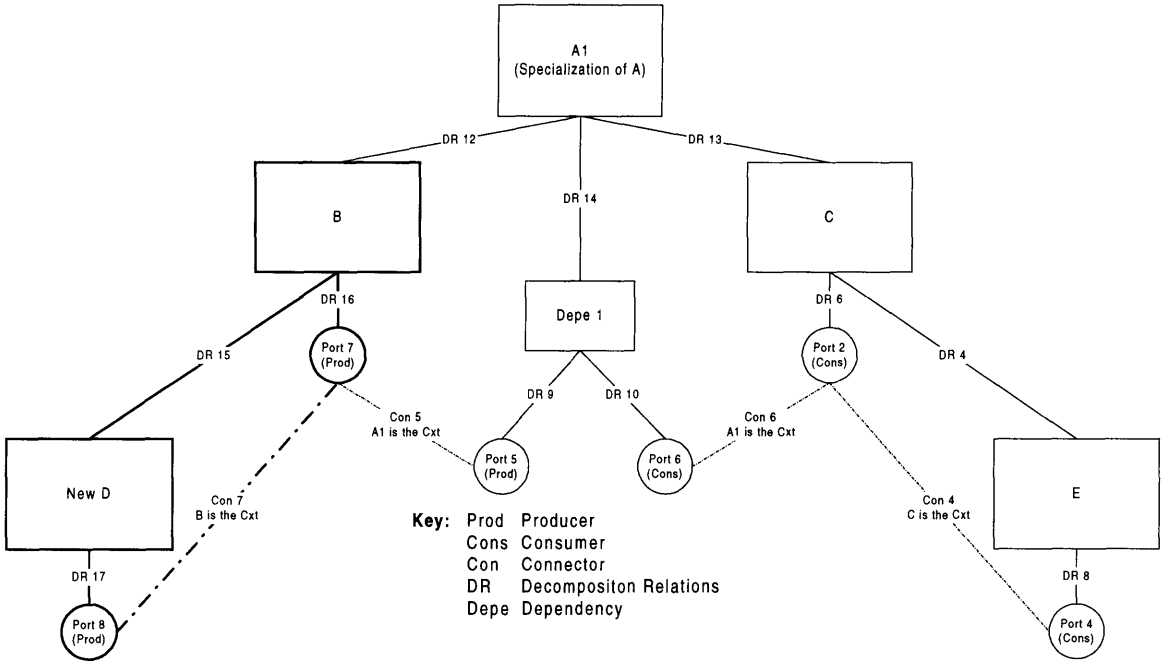


Figure 9c. Changes in the decomposition caused by modifying the activity *D* in *A1*'s decomposition context. Heavy lines represent the decomposition relations, connectors, activities, and ports as a result of this operation.

Figure 9c shows what happens when *D* is renamed to *New D* in the context of *A1*. First, the port of *D* is replaced by a new port. This illustrates the difference between the inheritance of ports and the inheritance of sub-activities and dependencies. Ports are not inherited by reference because ports are integral parts of the parent activity. It would not make much sense for two different activities *D* and *New D* to be using the same port to consume a resource. A new decomposition relation, connecting the port to its parent activity, is also created.

The creation of a new activity *New D* also causes the creation of a specialization of *B*. A new *B* is created because we have altered a sub-activity in the decomposition of *B*. Intuitively, altering an attribute (such as the name) of an activity is very similar to

changing the decomposition of an activity. The creation of a new *B* also results in the replacement of the port of the original *B* (*Port 1*)with a new port (*Port 8*). The connector *Con 3*, which existed in the context of *B*, is also replaced by a new connector *Con 7*.

Chapter 3

Goals of the New System

The two most commonly used implementations of the Process Handbook, the Windows version and the World Wide Web version, leave a lot to be desired (the system discussed in this thesis is currently in the development stage and as a result is not available to the users of the Process Handbook). The Windows version provides most of the functionality mentioned in chapter 2, but is becoming harder and harder to manage and extend due to the lack of modularity and abstraction in its design. The design of the server for the World Wide Web version is very similar to the design of the Windows version and as a result it suffers from nearly all the problems as the Windows version. Moreover, the Web version only provides a small subset of the functionality provided by the Windows version (please refer to Malone, et al., 1997 to read more about the functionality provided by the previous implementations of the Process Handbook). The new implementation of the Process Handbook aimed at fixing these problems. The following sections discuss the goals of the new system.

3.1 Easy Extensions to the Process Handbook Algorithms

It is becoming increasingly complicated to extend or change the algorithms in the current implementation. The main reason for this is a lack of modularity in the existing code. To understand the existing algorithms, one has to understand large amounts of code because there are very few low-level modules that can be used as black boxes. The problem is complicated further by the fact that there is a large overlap between the functionality being provided by different modules. Even the simplest changes to the existing functionality can become very time consuming and could lead to many bugs in the code. For example, if the developers of the Process Handbook decided that the names of all the bundles should be parenthesized, they would have to invest significant amounts of time and effort in making this small change. In a good system, this should be possible by simply changing the routine that reads the name from the database. All the other high-level modules that use the name information would then automatically use the modified

name. The problem with the current implementation is that many high level modules might be interacting with the database directly. Thus the developers would have to literally search the code for all the places where names are being read from the database. This can be a very time consuming and frustrating process for a code base as large as that of the Process Handbook.

Another, problem is that the routines being used in the Process Handbook have not been standardized. This also results from the lack of modularity. For example, until recently, the interaction with the underlying database was not very efficient. Some of the developers had used dynamic sets to read data from the database. Using dynamic sets, instead of snapshots of the database, lead to a significant decrease in efficiency. This inefficiency was only detected recently because the interaction with the database was embedded in very high-level routines. The inefficiency could have been detected much earlier if the modules for all the interaction with the database had been separated from the high-level algorithms. The new system aims at making it very easy to extend the current algorithms and standardize the low-level modules.

3.2 Easy Development of Clients for the Process Handbook

The true power of the Process Handbook comes from the semantic representation of the processes. The GUI (Graphical User Interface) should not limit the usefulness of the Process Handbook. The advanced users of the Process Handbook should be able to view and manipulate the information in the Process Handbook in any manner that they find useful. For example, if a consulting firm want to see all process as flow charts, they should be, with relative ease, able to add a GUI client that uses the Process Handbook methodology and display the processes as flow charts.

The current version of the Process Handbook makes it very hard to add new clients. The culprit, once again, is the overlap in functionality. The code that deals with the semantics of the Process Handbook is interspersed with the code that manages different GUI clients. For example, the code that adds specializations to entities is in the same modules that

display the boxes in the specialization viewer. To implement a very simple outline viewer for the specialization hierarchy (without any addition to the functionality), I had to identify and separate the code dealing with manipulating specializations from the display code. Thus it is not only necessary to understand the underlying semantic model, but also the implementation of the semantics. The advanced users of the new system should only have to understand the semantic model underlying the Process Handbook to add their own clients.

3.3 Abstraction of the Relational Database Design

The Process Handbook currently uses an MS-Access database (relational). The choice of the underlying database should not limit the Process Handbook's usefulness. For example, it should not matter if the underlying database is an MS-Access database or an Oracle database. Even if the underlying relational database is replaced with an object-oriented database, the semantics of the Process Handbook should not be impacted. Moreover, such a change in the underlying database should not require significant changes in the implementation of the interface (the GUI clients).

The current implementation of the Process Handbook lacks both these properties. The SQL (Structured Query Language) queries which interact with the underlying database are interspersed with the algorithms that deal with the semantics and interface of the Process Handbook. If the database was to be changed, the developers of the Process Handbook would have to locate all the places in the Process Handbook code where SQL queries are being made and make sure that the interaction with the database is not MS-Access specific (in the current version the interaction would actually have to be altered slightly to make it suitable for a different relational database). This would be an extremely tedious project.

Moreover, if we were to replace the underlying relational database with an object-oriented database, the code that manages the GUI might have to be changed significantly because some of the algorithms provide the inheritance functionality and manage the interface at

the same time. Replacing the relational database with an object-oriented one would significantly impact the inheritance algorithm which could also affect the current GUI clients. The new system should make the underlying database completely transparent and simplify the replacement of the existing database with other relational or object-oriented databases.

3.4 Database Integrity

It is very important to maintain the integrity of the underlying database because if the database gets corrupted, the corruption can be multiplied many times by the inheritance algorithms. A function as simple as the creation of a bad connector can lead to massive corruption of the database. For example, if a defective connector, one with common start and end points (i.e. a connector that connects a port to itself), were created in the decomposition of a generic activity, it would be inherited by hundreds, if not thousands, of specializations of that activity. Even if the connector is created in the decomposition of a very specialized activity, it would still corrupt the decomposition of all the activities that have that specialized activity as a sub-activity. To make things worse, if the database get corrupted, it can be very hard to fix. There are no routines in the current version of the Process Handbook that would detect or fix problems such as the looping connector. Thus the developers have to painstakingly locate the source of corruption, and in some cases manually fix the database.

The design of the current version of the Process Handbook makes it quite likely that a developer would corrupt the database accidentally because each new developer who joins the Process Handbook team has to re-program some of the basic functions. As these functions are programmed again and again, the chance of a mistake resulting in the corruption of the database increases. The new implementation should eliminate the need of this reinvention of the wheel and thus maintain the integrity of the database. The new system should also provide support for tracking down and fixing the corrupt parts of the database.

3.5 Full Internet Access to the Process Handbook

The increasing popularity of the Internet makes it necessary for the Process Handbook to be easily accessible over the Net. The version of the Process Handbook available over the Internet should also expose full functionality to the users. Providing full functionality for the Internet users is also important because despite the popularity of windows, many users still use the Macintosh Operating System and Unix (particularly in the academic community). The only way to make sure that these users can use the Process Handbook at their computer terminals is to allow access to the Process Handbook using the common Internet Protocols such as TCP/IP (Transmission Control Protocol/Internet Protocol) and HTTP (Hypertext Transfer Protocol).

The Web version of the Process Handbook does not give the users any editing privileges. The users who use the Handbook over the World Wide Web can only view the processes that are already a part of the Process Handbook database. This severely limits their interaction with the Process Handbook. One of the goals of the Process Handbook project is to gather information about as many different processes as possible. The usefulness of the Process Handbook increases with the number of processes that it contains. With the rapid growth of the Internet community (particularly the users with access to the World Wide Web), it is inevitable that the Web version of the Process Handbook will become more important than the Windows version. The new system should allow users to have full access to it over the Internet. Once the community of users, interacting with the Process Handbook over the Internet, has full access to the Process Handbook, the content in the Process Handbook will grow rapidly.

3.6 Easy Implementation of Security

Once the Process Handbook is fully accessible over the Internet, the number of users capable of adding new content and editing existing content would increase rapidly. This rise in the number of users would make it very important to implement security for the Process Handbook. The security features would protect the Handbook from malicious

users and allow the developers of the Process Handbook to keep track of the privileges of different users.

Security has not been implemented for the Windows version or the Web version of the Process Handbook. The Windows version can only be used by one user at a time. As a result, if the users are not using the same machines, they have to have their own copies of the database. A mistake made by one user does not affect other users. Moreover, only the developers and content providers for the Process Handbook have access to the registered database. As a result malicious users cannot damage the Process Handbook. The Web version does not give any user editing privileges at all. The new system would lead to full Internet access to the Process Handbook and should have advanced security features that can protect the content of the Process Handbook.

3.7 Forward Compatibility

One of the most remarkable characteristics of the software industry is the ever increasing number of new innovations. It should be easy to upgrade a system to a newer operating system or a newer version of the primary development language (Visual Basic for the Process Handbook) with relative ease. However, an upgrade of a large computer system can result in many management problems. For example, the developers have to support both the old and new versions of the code until all the users migrate to the new platform.

It is quite hard to upgrade the current version of the Process Handbook because of its monolithic design. Any upgrades result in changes to both the algorithms and the GUI. The complexity in upgrading is quite unavoidable with the current architecture of the Process Handbook (without a 'middleware' object API) because it cannot be solved by changing the structure of the code. Whereas most of the goals mentioned in the previous sections can be achieved by making introducing more modularity and abstraction into the Process Handbook design, forward compatibility requires a radical redesign and a new approach. The developers using the new system should be able to use it without any serious compatibility problems with the future software packages and operating systems.

3.8 Easy Interaction with other Software Packages

The users of a system should not be restricted to using any one software package to interact with or to extend the system. The advanced users should be able to use any software package to extend the system. For example, if a system is implemented using Visual Basic (the development language for the Process Handbook), the advanced users of that system should not be forced to learn or use Visual Basic. If such users could use the software packages that they were already familiar with (such as Visual C++) to develop extensions, they would save both time and effort. Moreover, in some cases it might be necessary to use other tools to interact with the system. For example, if an advanced user of the Process Handbook wants to use the power of an advanced drawing tool, such as Visio, with the Handbook, she should be able to do that with relative ease.

The current implementation of the Process Handbook does not allow the advanced users to use other software packages in conjunction with the Handbook. The current version of the Process Handbook uses Visual Basic and has a monolithic architecture. There is no easy means by which a user could access the information in the Process Handbook using other software packages. In order to use the Process Handbook functionality with a language other than Visual Basic, the user would have to first separate all the code that deals with the functionality (this is a huge task because of the lack of modularity in the code) and then translate the code to the other language (another difficult task). Thus, a lot of people who can make significant contributions to the Process Handbook might decide not to use it because of its incompatibility with other software packages. This problem also requires more than just modularity and abstraction in the code. The new implementation should solve this problem and allow easy interaction with other software packages.

3.9 A Small and Transparent Object API

The new system would allow the user to interact with the a server of Process Handbook objects by using an Object API. It is very important for this API to be concise. If the API is not concise enough, the new developers would have to learn the usage of long lists of

methods. Moreover, the API should not have multiple methods for using the same functionality. For example, the API should not have separate methods for getting the *Name* and *Contact* for an activity. Both *Name* and *Contact* are attributes of the activity and should be accessed by using a similar method. However, it should be clear that a API should not be made too small. Using techniques such as control-coupling (using switches in the arguments to a procedure to perform multiple tasks) in the methods for manipulating objects can reduce the size of the API, but it could lead to confusion.

The API should also be transparent to the users. This goal is related to the goal mentioned in section 3.3 (abstraction of the relational database design). The details of the implementation, including the language that we use to implement the server (Visual Basic), should be transparent to the users. Moreover, the structure of the underlying database should also be transparent to the users. Making the API transparent also has the added advantage that we could easily replace the database or change the implementation language because a transparent API is not dependent upon either the implementation details or the database schema..

3.10 Support for full Dependency and Attribute Inheritance

The current version of the Process Handbook does not fully support the inheritance of attributes and dependencies (see section 2.2.6 for the semantics of inheritance). As mentioned above, the attributes can be inherited in four different ways. The current version of the Process Handbook only supported two of the inheritance modes (inheritance with value and no inheritance at all). As the examples in section 2.2.6 show, the other two modes of attribute inheritance (inheritance with a default value and inheritance with no value) are also quite useful. The new system should provide attribute inheritance in all four modes.

The support for the inheritance of dependencies is even more important because dependencies form a critical part of the Process Handbook semantics. The current version of the Process Handbook does not support dependency inheritance as shown in

figures 9a, 9b,and 9c. The current version supports inheritance by copying (like the inheritance of Ports). This reduces the power of the system. For example, if the dependency in *A*'s decomposition (in figure 9a) was inherited by copying, the dependency in the decomposition of *A1* (in figure 9b) would be a new dependency. If the dependency in *A*'s decomposition was changed after this operation, the dependency in *A1*'s decomposition would not inherit the change. Thus, although *A1* inherits the decomposition from *A*, it is now different because the dependency in its decomposition does not inherit changes in *A*'s context. The new system should support inheritance for dependencies as explained in section 2.2.6.

3.11 Full Support for the Import and Export Processes

There are many representations of processes besides that used in the Process Handbook (such as IDEF0, Flow Charts, etc.). The Process Handbook should be able to exchange processes with applications that use these other representations. The inventor of the Process Handbook methodology (Prof. Thomas Malone), along with other scientists, has developed the specifications for the Process Interchange Format (PIF). The goal of developing PIF was to develop an interchange format to facilitate the automatic exchange process descriptions among a wide variety of business process modeling and support systems such as workflow software, flow charting tools, etc. (Malone, et al., 1996). The Process Handbook should be able to exchange data using the PIF specification.

The current implementation of the Process Handbook allows the users to import and export PIF processes. However, it does not give the user any control over the import process. For example, if a user wanted to import a PIF process generated by another application into the Process Handbook, the user cannot import only some of the activities in the decomposition of the process. This is because the current version of the Process Handbook does not allow the user to change the import procedure. The object API should not provide the user with a method to read and import a PIF file. Instead, it should only provide the user with the basic functionality (such as creating specializations, adding to the decomposition, etc.) and the user should decide how she wants to import the data.

However, this makes it necessary that the new system provides the user enough functionality to represent all the objects that make the PIF class hierarchy (this will be discussed in more detail in section 6.11) in the Process Handbook.

Chapter 4

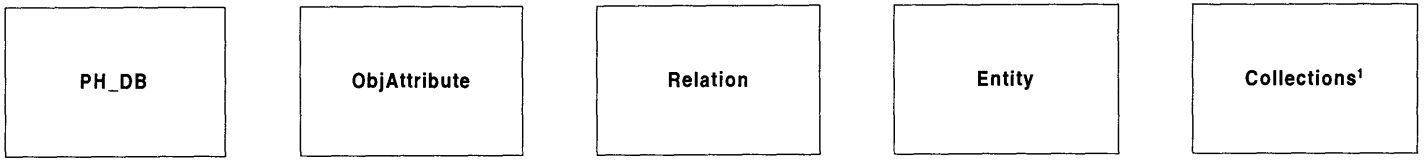
Overview of the Object API

4.1 Overview of the Objects and Methods

The object server exposes a total of seven first-class objects: PH_DB, Entity, Relation, ObjAttribute, Entities, Relations, and ObjAttributes. Figure 10 summarizes all the properties and methods for these objects (the detailed specifications for these properties and methods are attached as Appendix A). The PH_DB object allows the client to initialize the system and interact with the database directly. The client can then use the methods for Entity, Relation, and ObjAttribute objects to perform functions specific to the corresponding objects in the Process Handbook database. For example, the methods for the Entity object allow the user to access and modify the information or objects that are in some way related to the Entity object. Entities, Relations, and ObjAttributes are collections of Entity, Relation, and ObjAttribute objects respectively. Both Entity and Relation are typed objects i.e., there are sub-types within the Entity and Relation objects (these subtypes are discussed in more detail in chapter 5).

4.2 A Brief Description of How to Perform Basic Tasks Using the API

The following sub-sections describe the usage of the object API to perform the basic tasks in the Process Handbook. These sections are meant to familiarize a user of the new system with the API. Although details of the implementation and the specifications for the API is presented later in the thesis, I believe that the following sections can provide a user with enough information to perform basic functions in the Process Handbook. This is important because, to use Mark Klein's analogy, only providing implementation and specification information is equivalent to giving someone a box of tools without any instructions about how to use those tools to build a deck.



Methods:
OpenDB
GetRootEntity
GetEntity
GetObjAttribute
GetRelation
GetEntityByPIFID
GetAttriOwnerEntities
CloseDB

Properties:
Name
DataType
Value
Object
Inherited
ID
Deleted
OwnerObjectID
InheritedFromAttrID
InheritanceBehavior
DefaultValue

Methods:
SetValueInContext
SetObjectInContext
DeleteInContext
GetAttributeType
SetAttributeType

Properties:
Name
RelationType
StartID
EndID
Inherited
ID
Deleted
ContextID
InheritedFromRelID

Methods:
GetStartPath
GetEndPath
DeleteInContext

Methods:
GetAttribute
GetAttributes
AddNewAttribute
GetNavRelations
GetNavigatedFrom
AddNewNavRelation
GetSpecializations
GetGeneralizations
MoveToParent
RemoveFromParent
AdoptSpecialization
AddNewSpecialization
GetDecomposition
GetWhereUsed
AddDecomp
CreateConnector
GetConnectors
RemoveFromDecomp*
GetAttributeValue*
GetAttributeObject*
AddItemtoAttrCol*
RemoveItemFromAttrCol*
SetAttributeValue*
SetAttributeObject*
DeleteAttribute*
DeleteNavRelation*
ReplaceInDecomp*
DeleteConnector*

Methods
Count
GetItemByIndex
RemoveItemByIndex
AddItem

[1] These properties and methods apply to the three collection objects: ObjAttributes, Relations, and Entities.
* These methods are for efficiency in interaction with the server.

Figure 10. Properties and Methods for the First-Class Objects

4.2.1 Opening the Database and Getting Started

The *OpenDB* method for the PH_DB object opens the database. The method takes in the absolute path for the database file (the database file exists on the same machine as the object server). If the client tries to use any other method or property without opening the database, the server raises a Visual Basic exception. Once the database has been opened, the client can start the interaction with the server in four different ways. First, the client might get the root entity for the specialization hierarchy by using the *GetRootEntity* method. All other entities in the Process Handbook are specializations of this root entity. Second, the client might get a particular entity in the Process Handbook by using its ID (Identifier). The ID's of the entities in the Process Handbook might not be unique across different databases. Thus the client should know the entity's ID in the database that is being used by the server. This operation can be performed by using the *GetEntity* method. Third, the client might get an entity with a particular PIFID (Process Interchange Format ID) by using the *GetEntityByPIFID* method. The PIFID's of entities are unique across different databases; therefore the client can use PIFID for a particular entity from any Process Handbook database. Finally, the client can search for entities with a particular attribute by using the *GetAttriOwnerEntities* method. For example, if the client wants to get all the entities that have the word 'Financial' in their names, the client can use this approach. The client can also get an ObjAttribute object or a Relation object by using the *GetObjAttribute* method or the *GetRelation* method respectively. However, it would not make much sense to start the interaction with the database in this manner because both attributes and relations are attached to particular entities. A more structured approach would be to get an entity and then get the attributes and relations that are attached to that entity.

4.2.2 Navigating and Manipulating the Specialization Hierarchy

The specialization hierarchy forms a critical part of the Process Handbook. The API allows the client to easily navigate and manipulate the object hierarchy. The client can get the specializations or generalizations for a particular entity by using the *GetSpecializations* or *GetGeneralizations* method for that entity. The client can also add

specializations to an entity by using the *AddNewSpecialization* method for that entity. This is the only method that allows the client to add new entities to the Process Handbook. The API also allows the adoption of specializations. For example, if an activity *A* is the specialization of another activity *B* and the user wants *A* to become a specialization of both *B* and *C*, the client can use the method *AdoptSpecialization* method for entity *C*. However, if the client wants to move *A* from the specializations of *B* to the specializations of *C*, the client can use the *MoveToParent* method for entity *A*. The client can also remove an entity from the specializations of its parent by using the *RemoveFromParent* method for that entity. If the entity only has one parent, using the *RemoveFromParent* method would delete the entity.

4.2.3 Viewing and Changing the Decomposition of Entities

The viewing of information in the decomposition of an entity is quite simple. Figure 11 shows the decomposition of the activity *A* which can be used to illustrate the use of the methods dealing with the viewing and manipulation of the decomposition of entities.

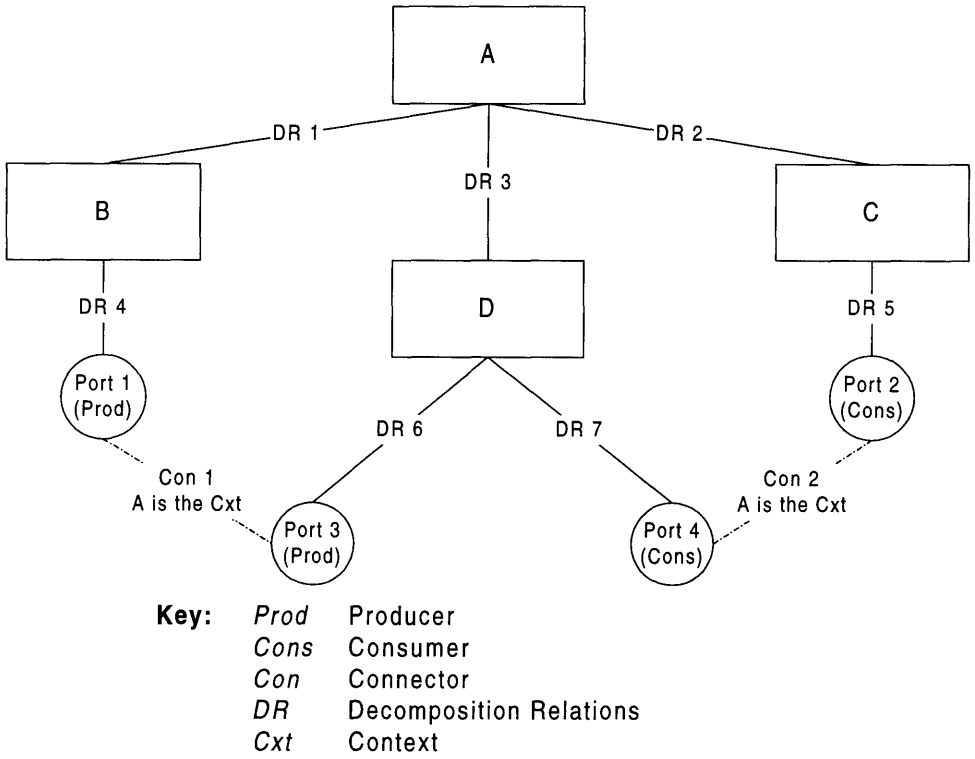


Figure 11. The decomposition of A.

The user can use the *GetDecomposition* method for entity *A* to get its decomposition. This method would return a collection of relations. This collection of relations would includes the relations that exit in the decomposition of the entity *A* (these consist of decomposition relations and connectors). The decomposition relations that are returned are the ones that start at the entity. Applying this method to *A* would return the decomposition relations *DR1*, *DR2*, and *DR3*. The connectors returned by this method are the one's that exist in entity's context (context for connectors is explained in 2.2.2). Applying the *GetConnectors* method to *A* would return the connectors *Con1* and *Con2*. Once the client gets the relations that describe the decomposition of this activity, the client can then use the relation object to find other entities that exist in the decomposition. For example, the client would use the *DR1* relation to get the activity *B*.

Adding or removing entities from the decomposition of an entity is a bit more complicated. The complication arises from the fact that the same entity can exist in the decomposition of many other entities (because of inheritance). The client has to pass in the decomposition path up to the context entity for all changes that are made in a decomposition context (this is discussed in more detail in section 4.2.4). For example, if the user wanted to remove *Port 1* from the decomposition of *B* in the context of the activity *A*, the client would also pass in an array with the ID's of the decomposition relations *DR4* and *DR1* to the *RemoveFromDecomp* method. The client would have to pass in the same information if the client wanted to add a port to the decomposition of *Port 1* in the context of activity *A*. by using the *AddDecomp* method.

4.2.4 Manipulating Attributes of Entities

As explained in section 2.2.4, the attributes of entities are divided into two groups: the system-generated attributes and the user defined attributes. All the system-generated attributes start with 'ph_.' The client cannot create an attribute whose name starts with these three characters. Figure 12 lists the system-generated attributes for the different sub-types of entity.

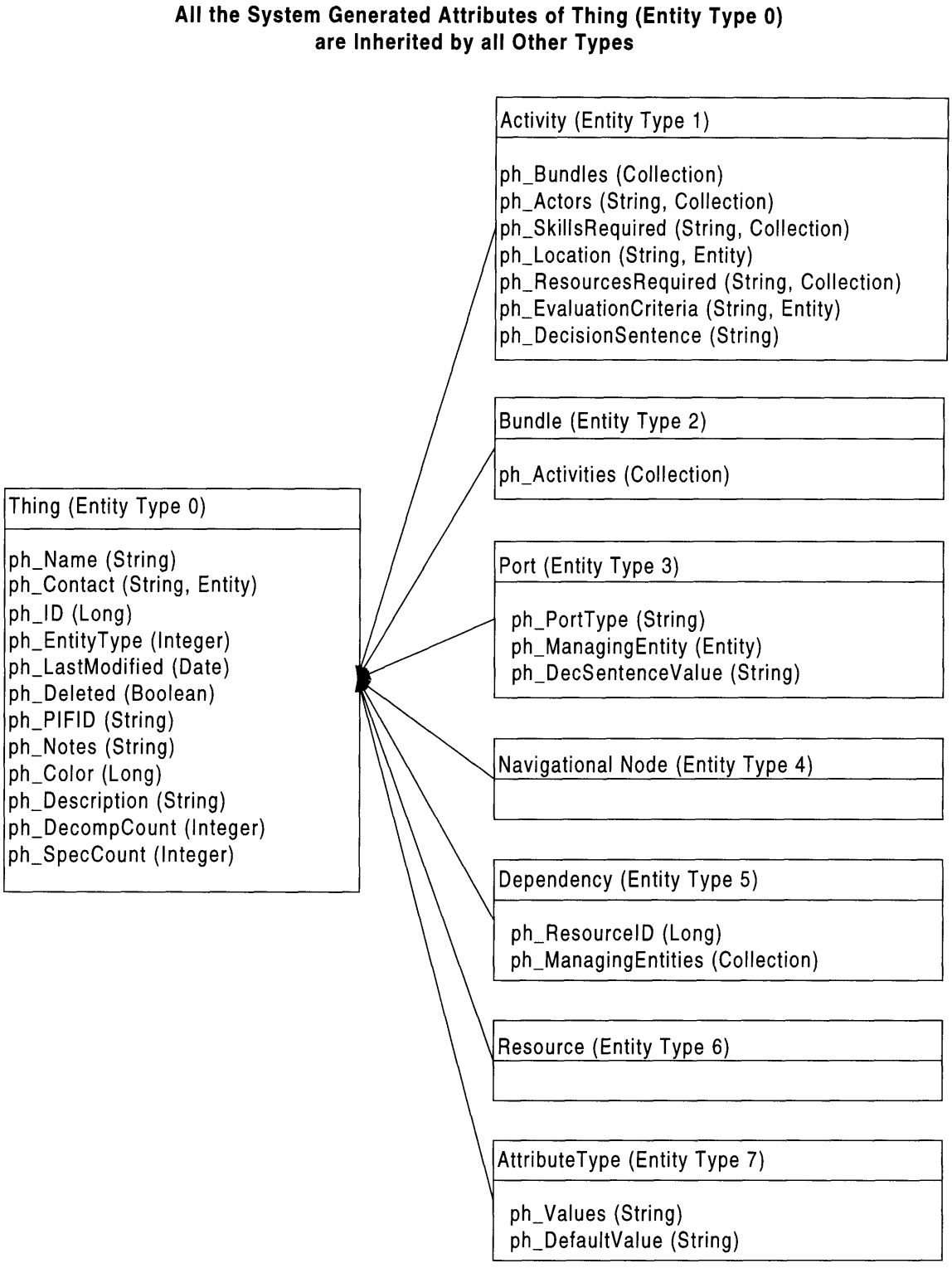


Figure 12. The system-generated attributes for the Entity object.

The client has to be able to read and change the attributes for all entities. For example, the client would need to read the attribute *ph_Name* to display the name of an entity to the user. The client can get the attributes of an entity in two different ways. First, the client can get a particular attribute of an entity by using the *GetAttribute* method for that entity. Second, the client can use the *GetAttributes* method for that entity. The second method returns a collection of all the attributes of that entity. The client can add new attributes for an entity by using the *AddNewAttribute* method for the entity. The client can also easily modify the value of the attribute (or the object stored in the attribute). However, the client has to specify the full decomposition context path (exactly as in the modification of the decomposition of an entity) to modify or add to the attributes of an entity in a particular decomposition context.

The bundles of an activity's specializations are treated as that activity's attributes. The client can obtain these bundles by getting the system-generated attribute named '*ph_Bundles*' for the activity. This attribute contains a collection of all the bundles. The client can get a collection of bundles by reading the *Object* property of the attribute and extract the bundles from the collection. Alternatively, the client can use the *RemoveItemFromAttrCol* method for the activity. The *RemoveItemFromAttrCol* method has been added to the API for efficiency reasons (this is discussed in more detail in chapter 5).

4.2.5 Setting a Managing Process for a Dependency

The Process Handbook uses the notion that coordination is the management of dependencies. Thus setting a managing process for a dependency will be a very common operation. This operation is slightly complicated in the cases where the dependency is not a single-producer, single -consumer dependency. The complication arises from the fact that the ports of a multiple-producer, multiple-consumer dependency have to be matched with the ports of the managing process. For example, in Figure 5b, the ports Port 7 and Port 8 have to matched with the ports Port 10 and Port 11.

The information about the managing entity for a port is stored in the *ph_ManagingEntity* attribute of the port; the information about the managing activity for a dependency, along with the dependencies created as a result of setting this managing activity, is stored in the *ph_ManagingEntities* attribute of the dependency. The client can set the managing process for the dependency *Depe 1* (shown in figure 5a) in two steps. First the client would have to set the managing ports for the ports of *Depe 1* i.e. the client would have to set the object inside the *ph_ManagingEntity* attribute for *Port 7*, *Port 8*, and *Port 9* to the entities *Port 10*, *Port 11*, and *Port 12* respectively. The user do this by using the *SetAttributeObject* method for the port entity. The order in which the client performs these three operations does not matter. Second, the client would have to add the Activity *M* to the collection inside the *ph_ManagingEntities* attribute of *Depe 1*. The dependencies, *Depe 2*, *Depe 3*, and *Depe 4* will only be created after the second step. These dependencies are automatically added to the collection in the *ph_ManagingEntities* collection. If the client wanted to set a managing process for a simple dependency, such as the one shown in figure 11, the client only has to set the managing process for the dependency *D*. In the simple (single-producer, single-consumer) case the ports are matched automatically.

4.2.6 Making Changes to Entities in Context

An sub-entity in the decomposition of an entity is considered to exist in the context of the entity. For example, in figure 11 (above), the activity *B* exists in the context of the activity *A*. Similarly, the dependency *D* exists in the context of the activity *A*, and Port 1 exists in the context of *B*. Decomposition relations and connectors can only exist in context of entities because they contain the information about the relationships between different entities in a decomposition context.

As mentioned in the previous sections, to make changes in the context of a decomposition, the client has to pass in the path in the decomposition context. The reason for this is that if the client changes an entity in a particular decomposition context, many of the other entities in the decomposition context might also get changed in the

process. This can be seen in Figures 9a, 9b, and 9c. Changing D in the context of A1 results in the creation of *B* (a specialization of the B in *A*'s decomposition), *Port 8*, *Port 7*, *DR 15*, *DR 16*, *DR 17*, and *Con 7*.

There are two possible approaches to deal with this situation. First, the server should notify the client of all the changes in the decomposition context. Second, the server should only notify the client that some of the objects have changed and let the client figure the changes out. We have included both the approaches in the design. Whenever the client makes a change in the context of a decomposition, the client is returned an array of tuples that contains the information about the objects that are modified in the decomposition context. The client is also returned a Boolean which the client can check to see if there are any changes in the decomposition. If there are no changes, the client can then just ignore the array of tuples. To see the detailed specifications for changes made in context see Appendix A.

Chapter 5

Implementation of the Object API

5.1 The Three-Tier Client/Server Architecture

The new implementation of the Process Handbook is designed as a three-tier client/server application. Figure 13 shows the three-tier architecture for the Process Handbook. This architecture consists of three separate layers (tiers): the database layer, the logic layer (which exposes Process Handbook objects), and the client layer (with GUI clients and other analysis tools).

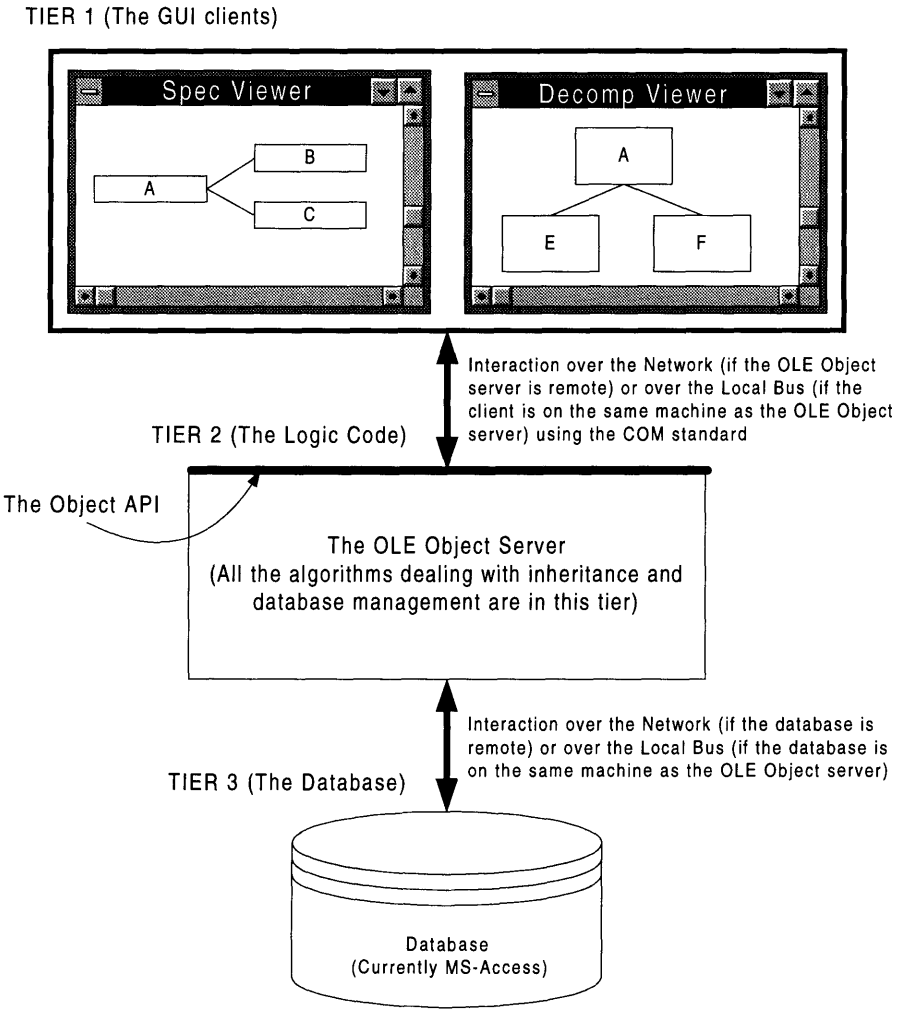


Figure 13. The Three-Tier Client/Server Design.

All of the information, used by the Process Handbook, is stored in the database which forms the bottom tier. The current implementation uses MS-Access to store the database. The new implementation will continue to use MS-Access because the database is still relatively small and does not require a more advanced tool (such as the Oracle database or the SQL server). The database only communicates with the object server (tier 2). The object layer acts as a client in its interaction with the database and gets data from the database by using SQL (Structured Query Language) queries.

The object server forms the middle layer. It contains the code for managing the database and for supporting the Process Handbook functionality. The object server exposes the object API to the client applications. The clients in tier 1 can only interact with the Process Handbook by using this server. Thus, the clients are prevented from performing illegal functions. The API abstracts the implementation of the algorithms and the storage device. Most importantly, the logic layer provides automatic support for inheritance. Whenever a client modifies the description of a process, this layer automatically creates/updates all the records in the database that are effected by these changes.

The object server exposes OLE objects and interacts with the clients in tier 1 by using Microsoft's COM (Component Object Model) standard for binary level interactions between applications. The main reason for this implementation was that the current Process Handbook software has been developed using Visual Basic which only supports OLE objects using the COM standard. Another important reason is that the COM standard has been developed by Microsoft and is currently the de facto standard for the software industry. All of the software tools developed by Microsoft will continue to support this standard. Therefore, using the COM standard ensures that the Process Handbook server will be fully forward compatible with the most popular applications. IBM's CORBA (Component Object Request Broker Architecture), which is the primary competitor of COM, is still in the development phase.

The viewers/editors for the Process Handbook form tier 1. These can either be GUI clients or other applications that interact with the Process Handbook. These clients can only interact with the Process Handbook by using the object API. These clients are very light-weight because they only have to deal with the representational issues and do not need to contain algorithms dealing with the implementation of the underlying semantics. As figure 12 shows, some typical clients might be viewers for viewing the specialization hierarchy or the decomposition of certain processes.

There are a number of different setups that we can use to implement the three-tier client/server system for the Process Handbook. The most general would be a classical three-tier architecture in which the three tiers exist on different machines and would interact with each other over the Internet or a local intranet. Alternatively, tier 2 and tier 3 can exist on the same machine and interact with remote tier 1 clients. In this case tier 2 and tier 3 would interact over the local bus of the machine and interact with tier 1 over the network. Another possible configuration is to have tier 1 and tier 2 on the same machine. In this case tier 1 and tier 2 interact over the local bus, while tier 2 interacts with tier 3 over the network. Yet another configuration is to have all three tiers on the same machine. In this case all the interactions between the tiers are over the local bus. We are using the last approach in the initial stages of implementations. However, once the implementation is completed, it should be very simple to move to any of the other configurations because Visual Basic makes it trivial to switch between these configurations.

5.1.1 The Advantages of the Three-Tier Architecture

The three-tier client/server architecture has five main advantages. First, the system is very robust because of the separation of the logic code and the tier 1 clients. For example, in the Process Handbook, the clients can no longer modify the underlying database directly. This prevents any inadvertent mistakes that might corrupt the database. Second, this decoupling abstracts the implementation of the complicated algorithms. Developers can add new clients to the Process Handbook without having to understand

the inheritance mechanism. Third, the components can be changed with relative ease. We could easily replace the Process Handbook database with another one as long as the interface between tier 2 and tier 3 remains unchanged. Fourth, this architecture makes the system more scaleable. Complicated algorithms could be added to the middle layer without affecting the other layers (if the object API remains unchanged). Finally, as mentioned in the previous section, the system can be configured in many different ways. For example, if large amounts of data are interchanged between tier 2 and tier 3, they could be installed on the same machine.

5.1.2 The Disadvantages of the Three-Tier Architecture

There are two disadvantages of the three-tier client/server architecture. First, and foremost, the performance of the three-tier system is worse than the monolithic system. This results from the decoupling of the top level (tier 1) clients from the database. All the data now has to go through the middle layer which slows down the system. If the three tiers are on different machines then high network latency or low network bandwidth can make the communications between the layers quite slow. Even if the tiers exist on the same machine, the system can be quite slow if the applications that from the different tiers are swapped in and out of RAM (Random Access Memory). Second, the client now has the responsibility of transferring all the information to the server. For example, in the new implementation of the Process Handbook the client has to provide the context information for making changes in the context of a decomposition. Though this is not a major problem, it can complicate the development of the clients.

5.1.3 Caching Between the Tiers

Currently we are developing the new version of the Process Handbook as a three-tier application in which all the tiers exists on the same machine. But, as mentioned above, it will be quite easy to separate the three tiers to different machines. If the three tiers are separated, it would be necessary to implement caches between the different layers. Although, network bandwidth might increase significantly in the future, at the present time the transfer of data over the network is slow. In the case of the Process Handbook, large amounts of data are exchanged between the tiers, making the system considerably

slow. Moreover, caching in the Process Handbook is complicated by the implementation of inheritance. Caching for the new system has been left for the future.

5.2 The First-Class Objects

The object server exposes a total of seven first-class objects: PH_DB, Entity, Relation, ObjAttribute, Entities, Relations, and ObjAttributes. The usage of these objects in performing the basic Process Handbook functions has already been discussed in chapter 4. A list of all the properties and methods for these objects has also been presented in that chapter (see figure 11). In this chapter, I will be discussing the implementation of these objects. I will also be discussing the sub-types of the Entity and Relation objects in detail.

5.2.1 The Reasons for Implementing Typed Objects

The Entity and Relation are typed objects i.e., there are sub-types within both these object (these subtypes are discussed in more detail in the following section). We considered both the advantages and disadvantages of implementing typed first class objects and decided that the advantages of this approach outweigh the disadvantage.

The main disadvantage of implementing sub-types within first-class object is that a number of errors are generated at run-time. Thus the programmer (who uses the API) has to be much more careful while programming. For example, the *AddDecomp* method for a port (entity type 6) can only take a port as argument. If both port and activity were a first-class objects, and a programmer tried to add an activity to the decomposition of a port, a compile-time error would be generated. In our current implementation, ports and activities are sub-types within the Entity object. Thus, if the programmer tried to add an activity to the decomposition of a port, the error would be discovered at run-time.

However, there are three main advantages of having sub-types within first-class objects. First, many first-class objects make it difficult for a programmer to get acquainted with the API quickly. Second, having sub-types within a single entity object makes it much

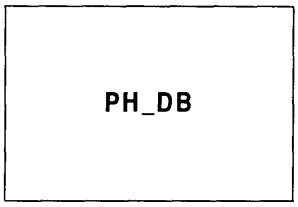
more efficient for the client to cache information about the entities being displayed in an editor. For example, the client might be showing activities, dependencies, and ports in the same editor. The most obvious way to cache all the information being displayed in such an editor would be to have this information in an array. If activities, ports, and dependencies were implemented as different first-class objects, the client would have to store them in an array of generic objects (because in Visual Basic, there is no other way to store different first-class objects in an array) and keep track of their types. The Entity object with it's sub-types makes it unnecessary for the client to keep track of the types because the client can store all the entities in an array of type Entity. The type of the displayed objects is stored within the Entity object. Third, and most importantly, if the developers of the Process Handbook decide to add a new entity type to the model, they would not have to change the API significantly. For example, if the developers were to add an Actor object, they would not have to change the methods applicable to the Entity object. They would only have to add documentation that specifies the semantics for the new object.

After discussing the pros and cons of having typed objects, we decided that having objects that have sub-types is preferable to having many different first-class objects. The sub-types within the Entity and the Relation objects are so similar that the methods applicable to them are almost identical. Thus the problem of generating run-time errors is mostly avoided. Moreover, the API documentation makes it very easy for the programmers to avoid these errors altogether (See Appendix A).

5.2.2 Interacting with the Database by Using the PH_DB Object

The PH_DB object is used to interact with the database directly. Figure 14 (overleaf) shows the methods for this object. The database is opened for exclusive use by a server because if multiple servers were to interact with the same database, the changes made by one server could effect the objects in another server. After opening the database by using the *OpenDB* method, the client can: get the root Entity; get an Entity, Relation or ObjAttribute object by using its ID; get an Entity by using its PIFID; or get entities with a

particular attribute. After the client finishes the interaction with the database, the client should execute the *CloseDB* method which executes database maintenance routines and closes the database.

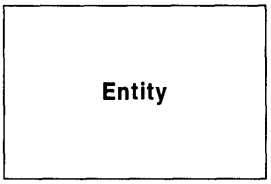


Methods:
OpenDB
GetRootEntity
GetEntity
GetObjAttribute
GetRelation
GetEntityByPIFID
GetAttriOwnerEntities
CloseDB

Figure 14. The methods for the PH_DB object.

5.2.3 Entity

The Entity object is the heart of the object server. There are eight sub-types within the Entity object: thing, activity, bundle, port, navigational node, dependency, resource, and attribute-type. All of these can be specialized and decomposed. This sets them apart from the relations and attributes, both of which can neither be specialized or decomposed. Moreover, the decomposition relations, connectors, and attributes only have any meaning in the context of an entity. The navigational relations (discussed later in the thesis) also have no meaning without the entities that form their end points.



Methods:

GetAttribute	AdoptSpecialization	AddItemtoAttrCol*
GetAttributes	AddNewSpecialization	RemoveItemFromAttrCol*
AddNewAttribute	GetDecomposition	SetAttributeValue*
GetNavRelations	GetWhereUsed	SetAttributeObject*
GetNavigatedFrom	AddDecomp	DeleteAttribute*
AddNewNavRelation	CreateConnector	DeleteNavRelation*
GetSpecializations	GetConnectors	ReplaceInDecomp*
GetGeneralizations	RemoveFromDecomp*	DeleteConnector*
MoveToParent	GetAttributeValue*	
RemoveFromParent	GetAttributeObject*	

Figure 15. The methods for the Entity object.

The methods for the entity object are applicable to all the sub-types. However, the arguments for some of the methods have some restrictions (see Appendix A). The methods for the Entity object are listed in Figure 15 (above).

Besides providing the methods necessary for the basic Process Handbook functions, we have provided a number of methods which increase the efficiency of the interactions between client applications and the object server (these methods are marked with a * in figure 15). This can especially be seen in the methods dealing with attributes of the entities. We initially considered implementing the most commonly accessed attributes (such as Name, ID, etc.)as properties of the Entity object. The advantages of implementing some of the attributes as properties would significantly reduce the number of API calls made by the client. For example, if the name of an entity had been implemented as a property of the entity, the client would only make one API call to get the name of a particular. If, on the other hand, the name of an entity had been implemented as an attribute of the entity, the user would first have to use the *GetAttribute* method to get the ObjAttribute object and then get the name by reading the *Value* property of that object. However, implementing some of the attributes as properties would make the API non-uniform and might confuse the developers who will be using the API. As a result, instead of implementing the commonly accessed attributes as properties, we have provided the *GetAttributeValue* method. This method allows the client to access the value of a particular attribute in a single API call. Similarly, the *SetAttributeValue* method makes it unnecessary to have two API calls for setting the value of an attribute. *GetAttributeObject*, *SetAttributeObject*, *AddItemtoAttrCol*, and *RemoveItemFromAttrCol* methods have been added to the API to make the interaction with the attributes, that contain objects or collections, more efficient. The usage for these methods is identical for system-generated (shown in Figure 12) and user defined attributes.

We have also added methods to make the API symmetric. More specifically, if the client can create an object by using the methods defined for the Entity object, the client should

also be able to delete that object by using these methods. This adds another group of methods, namely *DeleteConnector*, *DeleteNavRelation*, *RemoveFromDecomp*, and *DeleteAttribute*, whose functions can be performed in other ways. For example, instead of using the *DeleteConnector* method, the client could delete a connector by setting the *Delete* property for that connector to true.

Of the sub-types of the Entity object, Thing is the most general and has the entity type 0. The root of the Process Handbook specialization hierarchy has entity type 0. The system-generated attributes for the root are inherited by all of the entities. The first level of specialization hierarchy is illustrated in figure 16. The behavior and system-generated attributes for the different entity types are different and require some discussion.

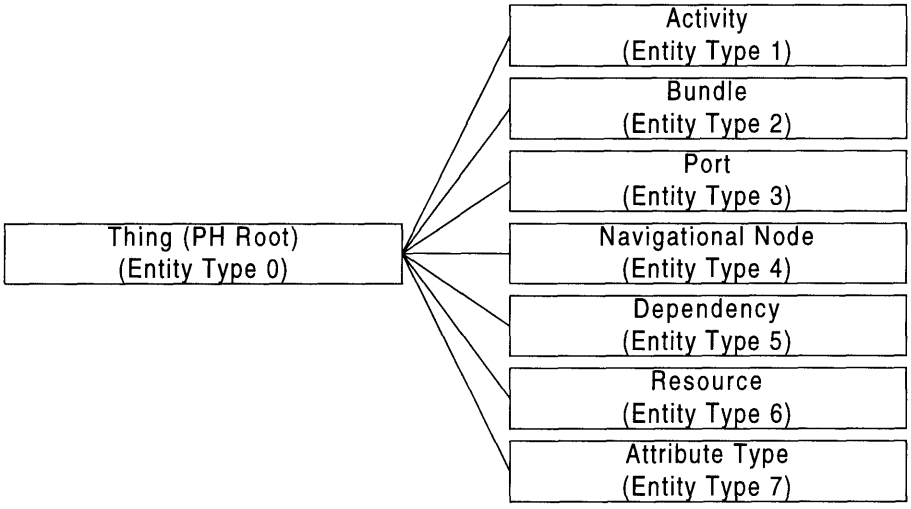


Figure 16. The first level of the specialization hierarchy.

(i) Thing

Thing is a general entity. The main reason for having a Thing type is to allow the easy addition of new types to the Process Handbook. For example, if the client wanted to add an Actor object to the Process Handbook, it would add that object as a specialization of Thing. If the client, at some later stage, decided that the new object should actually be an activity, the client could use the *MoveToParent* method to move that object to be a specialization of an activity. Thus, if an object is moved from the specializations of a

thing to a specializations of another sub-type, the object is automatically changed to that type.

Table 1 shows the system-defined attributes for the Thing object (all the entities in the Process Handbook have these system-generated attributes). Most of the attributes are self-explanatory. The attributes *ph_DecompCount* and *ph_SpecCount* allow the client to find out the number of entities in the decomposition of this entity and the number of specializations for this entity respectively. The attribute *ph_Color* allows the client to specify the display color for a particular entity.

Attribute Name	*Value Data Type*
ph_Name	String
ph_Contact	String, Entity
ph_ID	Long
ph_EntityType	Integer
ph_LastModified	Date
ph_Deleted	Boolean
ph_PIFID	String
ph_Notes	String
ph_Color	Long
ph_Description	String
ph_DecompCount	Integer
ph_SpecCount	Integer

Table 1. The system-generated attributes for the entity of type Thing.

A thing can have both specializations and decomposition. All specializations of a thing inherit its decomposition and its attributes. The decomposition of a thing can only contain other things. However, once a thing is moved to be a specialization of one of the other entity types, it starts behaving like the latter entity type.

(ii) Activity

Activities have already been explained in some detail in section 2.2 so I will not dwell upon their semantics. However, all of the system-generated attributes of activities were not discussed in that section. Table 2 lists the system-generated attributes of activity

(besides the ones that have already been explained in section (i) above). The attribute *ph_Bundles* has already been explained in the section 4.2.4. The attributes *ph_Actors*, *ph_SkillsRequired*, *ph_Location*, *ph_ResourcesRequired*, and *ph_EvaluationCriteria* store information about the actors that perform the activity, the skills required for performing the activity, the location where the activity is performed, the resources required to perform the activity, and the evaluation criteria for the activity respectively. The user can store both strings and Process Handbook entities in *ph_Actors*, *ph_Location*, and *ph_ResourcesRequired*. The reason for allowing both strings and entities is that it might not be possible to represent the values of these attributes solely in terms of the entities in the Process Handbook.

Attribute Name	*Value Data Type*
ph_Bundles	Collection of Entities
ph_Actors	String, Collection of Entities
ph_SkillsRequired	String, Collection of Entities
ph_Location	String, Entity
ph_ResourcesRequired	String, Collection of Entities
ph_EvaluationCriteria	String, Entity
ph_DecisionSentence	String

Table 2. The system-defined attributes for the entity of type activity.

The attribute *ph_DecisionSentence* is only meaningful for the activities that signify a decision process. The value of this attribute is an expression that can be evaluated to a string (for example "True" or "False"). This result can then be used to choose between alternatives. A decision activity can only be used to pick multiple alternatives. The semantic structure for the decision activities is discussed in more detail in section (iv) below.

(iii) Bundle

Bundles have already been discussed in considerable detail in section 2.2.5. Bundles only have one system generated attribute named *ph_Activities*. This attribute contains the collection of all the activities that are in the bundle. This attribute can be manipulated in

exactly the same manner as the *ph_Bundles* attribute for activities (see section 4.2.4). The tradeoff matrices for a bundle are stored in its user generated attributes. This gives the client full liberty in representing the information pertaining to bundles.

Bundles can be specialized or decomposed. If a bundle is specialized, its specialization would inherit all the activities in the bundle. At this time, the system allows bundles to be decomposed into other bundles, but the semantic uses of such a decomposition are not very clear. At the time of the writing of this thesis, the semantic model for the decomposition of bundles is still being discussed and developed.

(iv) Port

Ports have also been discussed in section 2.2.2. Table 3 lists the system-generated attributes for ports. The attribute *ph_PortType* specifies the type of a port. All ports are either "producer", "consumer", or "untyped". The attribute *ph_ManagingEntity* is used to set the managing entity (a port of a managing activity) for the port. This is part of the process that sets a managing activity for a dependency (see section 4.2.5).

Attribute Name	*Value Data Type*
ph_PortType	String
ph_ManagingEntity	Entity
ph_DecSentenceValue	String

Table 3. The system-generated attributes of ports.

The attribute *ph_DecSentenceValue* works along with the *ph_DecisionSentence* attribute for a parent activity to pick alternatives. Figures 17a and 17b illustrate an example where having the *ph_DecisionSentence* and *ph_DecSentenceValue* attributes lends power to the Process Handbook. Figure 17a (overleaf) shows a process which describes the execution of two activities. The activity A gets executed first. Then depending upon the value of the decision made by the activity D, either B or C will be executed.

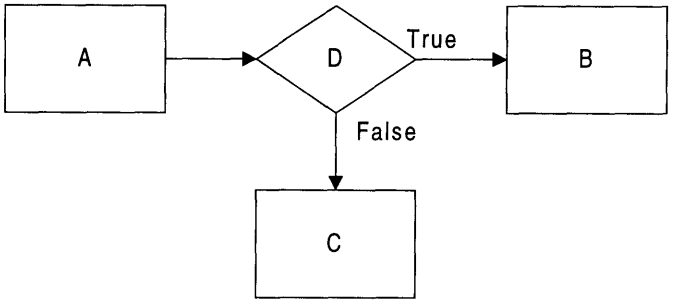


Figure 17a. A process that includes a decision D.

Figure 17b illustrates how the attributes mentioned above can be used to represent this process. The attribute *ph_DecisionSentence*, belonging to the activity D, has as its value an expression that evaluates to the string "True" or the string "False." (The expression stored in ph_DecisionSentence can evaluate to any string and is not limited to "True" and "False"). If the expression evaluates to "True", *Port 3* is used, otherwise *Port 4* is used. The arrows in figure 17a are converted to flow dependencies because activities can only interact with each other through dependencies.

All ports can be specialized and decomposed. The specialization of a port inherits its attributes and decomposition. A port can only have other ports in its decomposition. The notion of the decomposition of ports is quite powerful, especially for the ports that are untyped. For example, if an untyped port is being used for a bi-directional dependency, it can be decomposed into a producer port and a consumer port.

(v) Navigational Node

Navigational nodes can be thought of as folders for storing references to similar activities. Navigational nodes are related to the entities through navigational relations (explained in section 5.2.4). The concept of navigational nodes is very similar to that of folders of bookmarks in a Web browser. Each folder (navigational node) of bookmarks can contain bookmarks (references to entities) and other folders (navigational nodes in the decomposition).

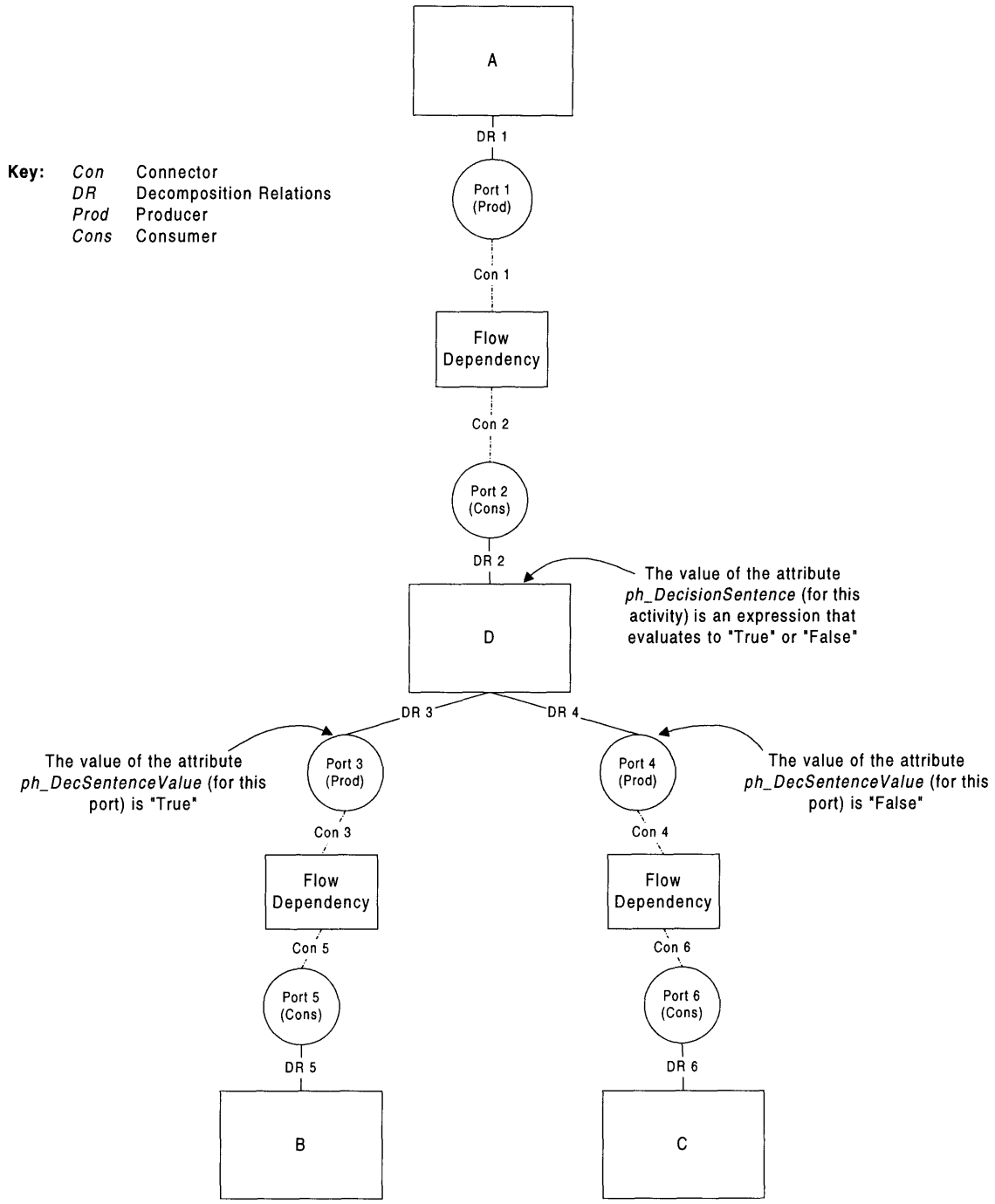


Figure 17b. The semantic structure of the process shown in figure 17a.

Figure 18 (below) shows three navigational nodes *NA*, *NB*, and *NC* which contain references to the entities *1* through *9* (which are not navigational nodes). The navigational node *NA* contains references to three entities and can be decomposed into

two other navigational nodes *NB* and *NC*, each of which also contains references to three entities.

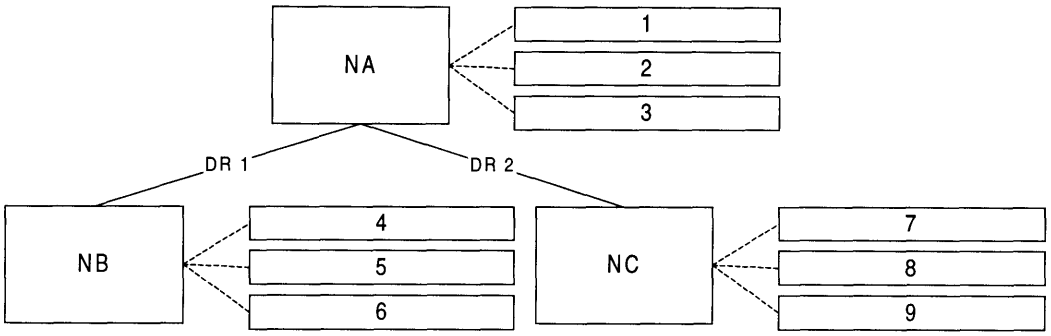


Figure 19. An illustration of the relationship between navigational nodes and other entities.

Navigational nodes can be decomposed and specialized. The specializations of a navigational node inherit its decomposition, attributes, and navigational relations. A navigational node can only have other navigational nodes in its decomposition.

(vi) Dependency

Dependencies have also been discussed quite extensively in previous sections. Table 4 lists the system-generated attributes of a dependency. The attribute *ph_ResourceID* is used to store information about the resource that is associated with this dependency. A dependency can only have one resource associated with it at any given time. The attribute *ph_ManagingEntities* has already been discussed in detail in section 2.4.5.

Attribute Name	*Value Data Type*
ph_ResourceID	Long
ph_ManagingEntities	Collection of Entities

Table 4. The system-generated attributes of dependencies.

Dependencies can be specialized and decomposed. The specializations of a dependency inherit its decomposition and its attributes. Dependencies can be decomposed into ports

and other dependencies. The concept of decomposing dependencies into other dependencies also adds a lot of power to the Process Handbook. For example, a flow dependency can be decomposed into a prerequisite dependency, a accessibility dependency, and a usability dependency.

(vii) Resource

A resource is an object which can modify an activity or which can be produced, modified, or consumed by an activity. A resource can get transferred from one activity to anther (flow dependency), shared among many multiple activities (share dependency), or produced by multiple activities (fit dependency).

Resources can be specialized and decomposed. The specialization of a resource inherits its attributes and decomposition. A resource can only be decomposed into other resources. For example, a complex resource such as a car can be decomposed into its components.

(viii) Attribute-Type

Attribute-types only add representational power to the Process Handbook. An attribute-type can be thought of as being a set of possible values that an attribute can have. For example, if a resource has an attribute called *Intelligence* and this attribute can only have three values, namely low, medium, or high, then the set of low, medium, and high would form an attribute-type.

Table 5 (below) shows the system-generated attributes of the attribute-type object. The attribute *ph_Values* contains a comma-delimited string of all the values that are contained in the attribute-type. In the example mentioned above, the attribute *ph_Values* would contain the string "low, medium, high." The attribute *ph_DefaultValue* contains the default value for this attribute-type. For example, if the attribute *ph_DefaultValue* for the above mentioned attribute-type contained "medium" and the user set this attribute-type

for an ObjAttribute object, the value for the ObjAttribute will automatically be set to "medium" (ObjAttribute objects are discussed in more detail below).

Attribute Name	*Value Data Type*
ph_Values	String
ph_DefaultValue	String

Table 5. The system-generated attributes of attribute-types.

Attribute-types can also be specialized and decomposed. The specializations of an attribute-type inherit its attributes and decomposition. An attribute-type can have other attribute-types in its decomposition. The decomposition of attribute-types is useful when possible values, stored in an attribute-type, are records. In this case, the attribute-types in the decomposition would contain the set of possible values for individual fields in the record.

5.2.4 Relation

A Relation object relates two Process Handbook entities. There are three sub-types within the relation object: decomposition relation, connector, and navigational relation. Figure 20 shows the properties and methods for the relation object.

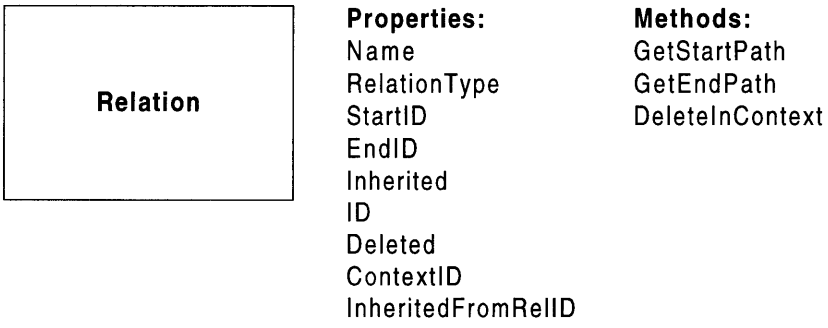


Figure 20. The properties and methods for the Relation object.

The *Inherited* property specifies whether the relation is inherited or not. This property is primarily used for decomposition relations and connectors in a decomposition context. Figures 21a and 21b (reproductions of figures 9a and 9b with minor modifications) illustrate the usefulness of this property. This property would be true for the decomposition relations *DR 12*, *DR 13*, and *DR 14*, and the connectors *Con 5* and *Con 6* in figure 21b. Whenever, a relation is inherited, the property *InheritedFromRelID* for the inherited relation contains the ID of the original relation. In the above example, the *InheritedFromRelID* property for the relations *DR 12*, *DR 13*, *DR 14*, *Con 5*, and *Con 6* would contain 100, 200, 300, 400, and 500 respectively.

The *StartID* and *EndID* properties contain the ID's of the entities that form the two end points for the relation. Only navigational relations and decomposition relations are directional. The connectors do not have any directional information in them because the direction of a connector can be deduced by reading the *ph_PortType* property for the ports that the connector connects.

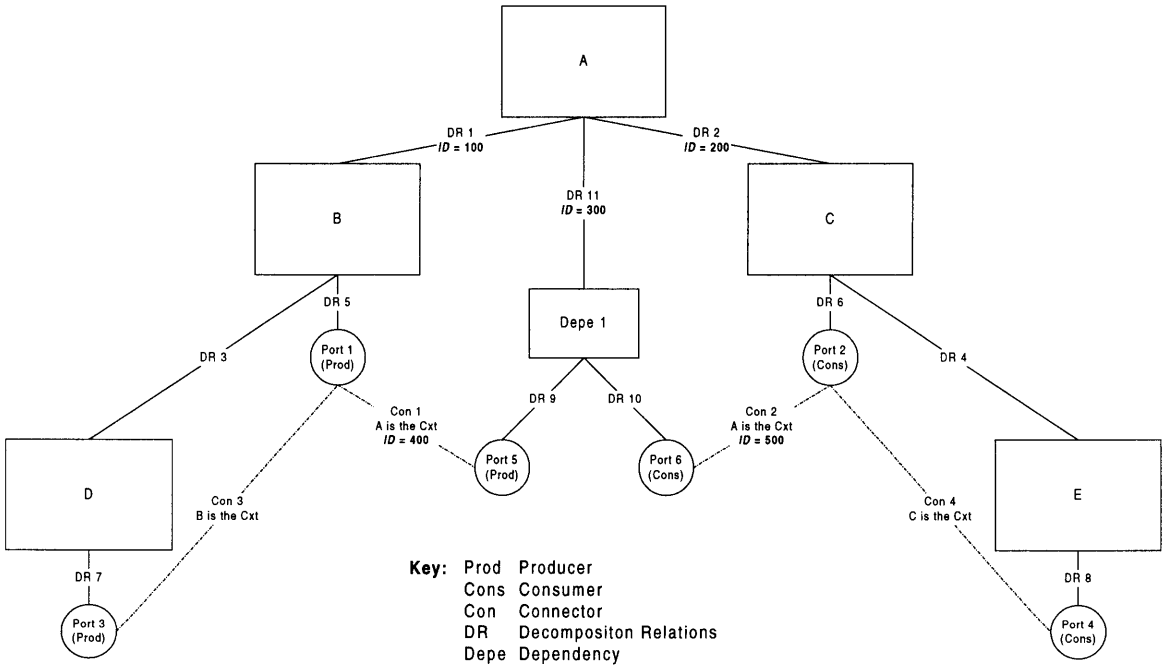


Figure 21a. The two level decomposition of the activity *A*.

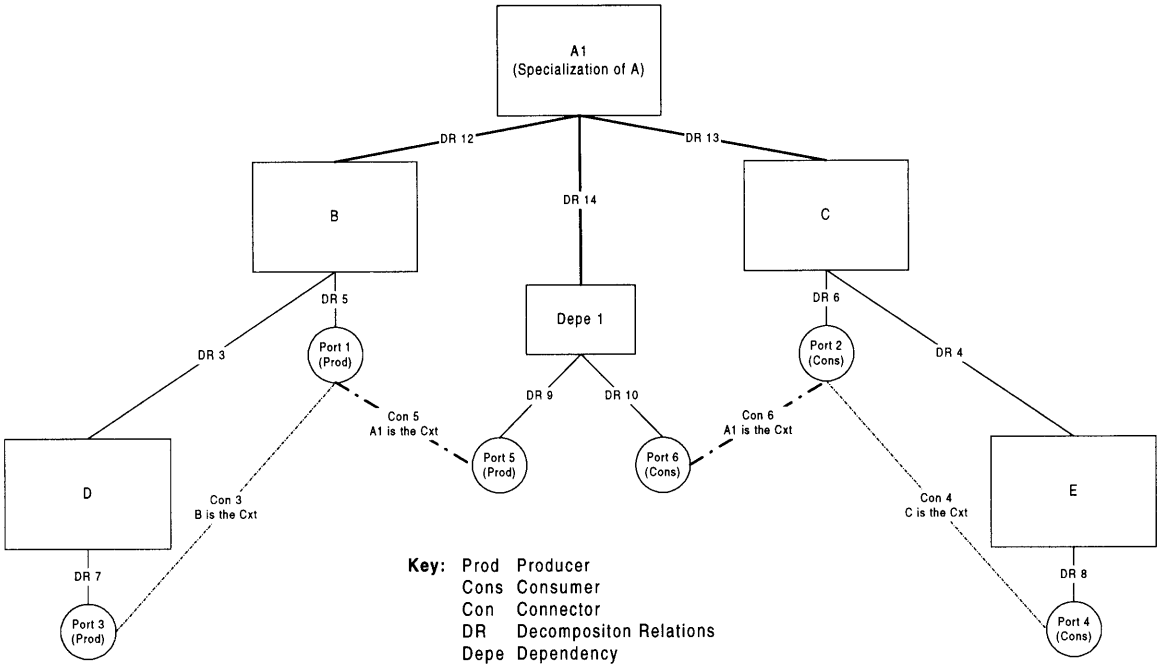


Figure 21b. *A1* is a specialization of *A*. It inherits the decomposition of *A*. Heavy lines represent the newly created decomposition relations.

The *ContextID* property contains the ID of the entity in whose context the relation exists. This property is only useful for the connectors. As discussed in chapter 2, all connectors exist in context of entities. For decomposition relations and navigational relations, this property has the same value as the *StartID* property.

The *GetStartPath* and *GetEndPath* methods apply to connectors and return arrays of decomposition relation ID's that define the paths from the end ports of the connector to the context entity. These methods are useful when an entity exists twice in a single decomposition. Figure 22 (overleaf) shows a decomposition where the activity *B* exists twice in the decomposition of *A*. If the client only read the *StartID* property for the connector *Con 1*, the client would get the ID of *Port 4*. But this information is not enough to infer the actual structure of the decomposition. However, once the client uses the method *GetStartPath* for this connector, the client would get an array containing the ID's 6 and 1 (the ID's of *DR 6* and *DR 1* respectively). The information about the complete path allows the client to understand the structure of the decomposition completely.

The method *DeleteInContext* is required to delete decomposition relations and connectors because the client has to provide the arrays that contain the information about the context (as explained in sections 4.2.3 and 4.2.4).

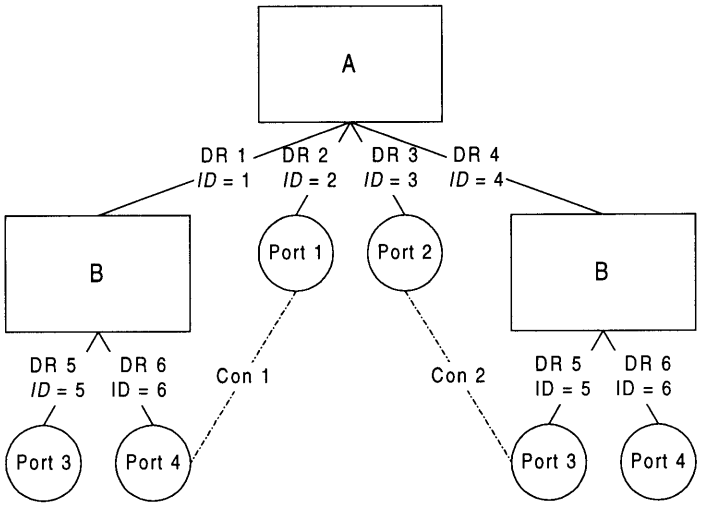


Figure 22. An example of a decomposition with multiple instances of the same activity.

5.2.5 ObjAttribute

Each ObjAttribute object represents an attribute belonging to a Process Handbook entity. Attributes are discussed in detail in section 2.2.4. The API interface for the system-generated attributes and the user-defined attributes is identical. The only difference between these two kinds of attributes is that the names for the system generated attributes start with 'ph_'. The users are not allowed to add any attributes whose names start with these characters.

Figure 23 (overleaf) illustrates the properties and methods for the ObjAttribute object. The *Name*, *Inherited, InhertiedFromAttrID, ID*, and *Deleted* properties are self-explanatory. The value of the *OwnerObjectID* property is the ID of the entity that this attribute belongs to. The *InheritanceBehavior* property specifies the manner in which this attribute is inherited (see section 2.2.6 for detail about inheritance of attributes).

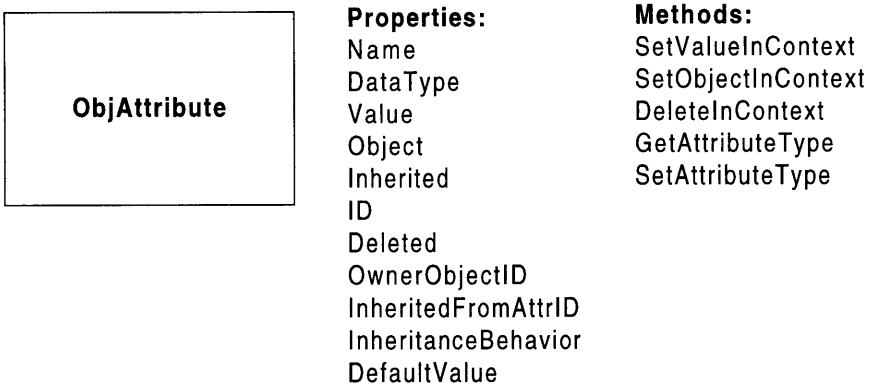


Figure 23. The properties and methods for the ObjAttribute object.

The *DataType* property specifies the type of value stored in this attribute. This can either be an ordinary data type, such as string, integer, long etc., an object, or a string and an object. If the value is of an ordinary data type, this value is stored in the *Value* property of the ObjAttribute object. However, if the value is an object, it is stored in the *Object* property. The client can store one of three different kinds of objects inside an attribute: an OLE object, a Process Handbook entity, or a collection of Process Handbook entities. When the attribute contains both a value and an object, both *Value* and *Object* slots are filled.

The *DefaultValue* property specifies the default value for this attribute. This property is only used for the attributes which are inherited with their default values.

The methods *SetValueInContext*, *SetObjectInContext*, and *DeleteInContext* are used to set the value of, set the object in, and delete an attribute (respectively) in the context of a decomposition. The methods *GetAttributeType* and *SetAttributeType* get and set (respectively) the attribute-type entity that contains set of possible values for this attribute. When the client sets the attribute-type for a particular ObjAttribute object, the value of that attribute is automatically set to the default value of the attribute-type entity (stored in the *ph_DefaultValue* attribute of the attribute-type entity).

5.2.6 Collections

The server exposes three collections to the user: the collection of Entity objects, the collection of Relation objects, and the collection of ObjAttribute objects. These collections can be manipulated by using the methods listed in figure 24. The methods available for the collection have very simple behavior. The *Count* method can be used to find out the number of entities in the collection. The *GetItemByIndex* method returns a member of the collection without removing it from the collection, whereas the *RemoveItemByIndex* method returns the member and removes it from the collection. The *AddItem* method can be used by the client to add another member to the collection.

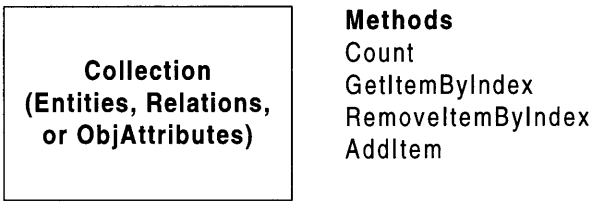


Figure 23. The methods for the collections.

5.3 Database Schema

Figure 24 (overleaf) shows the schema for the relational database that is used by the object server. The schema separates the entities, relations, and attributes into separate tables. However, there are two characteristics of the schema that should be mentioned. First, the relationships between an entity and its specializations are not included in the *Relation* table. The main reason for this is efficiency in fetching the specializations of an entity (we think that this will be the one of the most commonly used functions in the Process Handbook).

Second, the attributes for an entity are stored in three different manners in the database. The most commonly accessed attributes for an entity, such as name, contact, etc., are stored in the *Entity* table. Thus whenever the object server makes a database query to get an entity, the server can also get these attributes in the same query. This makes the

interaction between Tier 2 and Tier 3 very efficient. Some other attributes that are not accessed as often, such as the managing entity ID for a dependency, are stored in the *Relation* table. This also makes the database queries for these attributes considerably faster because the *Attributes* table is much larger than any of the other tables; as a result searching through the *Attributes* is about an order of magnitude slower than searching through any other table. The least commonly accessed attributes are stored in the Attributes table.

Note: In figure 24, the tables whose names end with '_1' do not physically exist in the database. They have been added to the figure to show self-reference in the *Entity*, *Relation*, and *Attributes* tables.

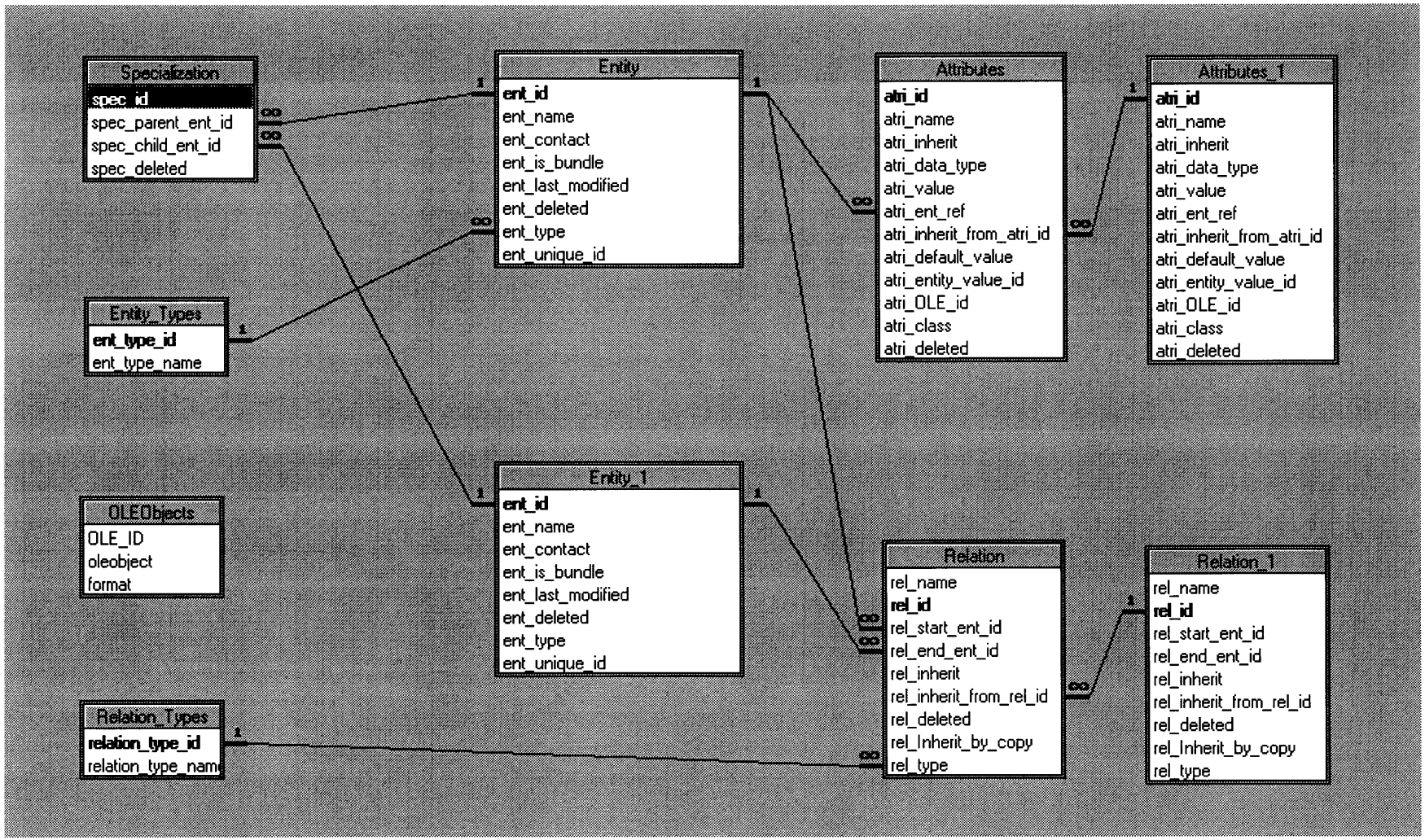


Figure 24. The schema of the current relational database used by the Process Handbook.

Chapter 6

Evaluation of the Achievement of Goals

The new three-tier client/server architecture has made it possible for us to achieve the goals defined in chapter 3. Modularity and abstraction are the attributes of the new system that have made it possible for us to achieve most of the goals. Other design decisions, such as using OLE and the COM standard, contribute towards the achievement of the other goals. The following sections describe the success in achieving each of the goals listed in chapter 3.

6.1 Easy Extensions to the Process Handbook Algorithms

The new implementation is very modular and uses object-oriented design. All the methods that can be performed on the first-class objects are completely encapsulated. This makes it unnecessary for developers to understand the implementation of these methods. A developer only has to understand the specifications for these methods and can use them as black-boxes. Thus the new implementation completely avoids the 'reinvention of the wheel' problem.

Moreover, as long as the new extensions do not affect the object API, their impact on the tier 1 clients, using the object server, would be minimal. This gives the Process Handbook developers considerable flexibility in changing the algorithms. For example, the algorithms that interact with the database can be made more efficient without worrying about introducing bugs into the display algorithms.

6.2 Easy Development of Clients for the Process Handbook

Once the logic of the object model in the Process Handbook is abstracted away, building new clients for the Process Handbook would become considerably easier. The main problem in the current monolithic implementation is that developing clients involves writing considerable amounts of code that implements the core functionality of the Process Handbook. Once the server is completely functional and bug-free, the developers

working on creating clients only have to concerned with the client design issues (such as GUI representation etc.). This would make it considerably easier and quicker to implement new clients.

The difference in the ease of development has already been seen in the initial tests of the server. Implementing a list viewer for the specialization hierarchy using the object server required about one-fourth of the time required to implement a similar list viewer for the current monolithic model.

6.3 Abstraction of the Relational Database Design

The database schema has been abstracted at two different levels. First, the design is hidden from the users of the Process Handbook. The best example of this can be seen in the implementation of attributes in the new system. As discussed in section 5.3, the attributes are stored in either the *Entity* table, the *Relation* table, or the *Attributes* table depending upon the frequency with which they are accessed. These schema details are completely hidden from the clients. For a client using the object API, there is no difference between the *ph_Name*, *ph_ResourceID*, and *ph_Description* attributes.

Even within the server, the database details have been abstracted. The server interact with the database through a private database object (this object is not exposed through the API). Thus the developers of the Process Handbook can easily replace the underlying relational database as long as the interface for the database object is not changed. The database can even be replaced by an object-oriented database with considerable ease.

6.4 Database Integrity

The modularity and abstraction introduced by the new design will ensure database integrity. The clients interacting with the object server can only use the object API to alter the contents of the database. As long as the server code is bug-free, the client cannot corrupt the database. At this stage, we are making the assumption that the database will

not be corrupted by any routine in the server. As a result, we have not included any routines in the server that regularly check the database for any inconsistencies.

6.5 Full Internet Access to the Process Handbook

The new implementation will allow remote uses to interact with the Process Handbook in two way. First, users could interact with the Handbook using the World Wide Web. The current Web server for the Process Handbook does not provide any editing privileges to the Web clients. A new Web server can easily be developed which would use the OLE objects exposed by the object server to provide full Process Handbook functionality to remote users.

Remote users can also interact with the object server using tier 1 clients developed for the Windows 95 or Windows NT operating systems. The object server has been designed using Visual Basic which supports direct TCP/IP (Transmission Control Protocol/Internet Protocol) requests by clients running on remote machines. It is relatively simple to build tier 1 clients (in Visual Basic) that can interact with a remote OLE server.

6.6 Easy Implementation of Security

Since multiple users can now use the Process Handbook (over the network), it has become very important to introduce security features into the Process Handbook design. Clavin Yuen has designed an optimal and flexible access control policy which regulates the usage of the Process Handbook and increases the flexibility for collaboration in a multi-user Handbook (for more detail for the access control policy, see Yuen, 1997).

At the time of writing of this thesis, the access control policy has not been merged with the object API. Once security features are added to the object server, the object API will change. However, the resulting changes would not impact the design of the first-class objects or the database schema very much. Moreover, the semantic model of the Process Handbook described in this thesis will remain unchanged.

6.7 Forward Compatibility

We decided to implement the object server using OLE objects. This makes the Process Handbook forward compatible. OLE has become an industry standard and will continue to be supported by new development tools that come to the market in the future. Even if the clients (tier 1) are programmed in future versions of Visual Basic or Visual C++, the object server would not need to be changed. Currently, OLE objects are only supported by applications developed for the Windows operating system. However, some software firms are working on tools that would enable Macintosh Operating System and Solaris (operating system for Sun workstations) to support OLE objects.

6.8 Easy Interaction with other Software Packages

Using OLE objects also gives the users a lot more flexibility in viewing and using the data that is obtained from the Process Handbook server. In the past we have had to make extensions to the Process Handbook to enable exporting the specializations and the decomposition of an entity as MS-Excel spreadsheets. The new implementation makes such operations very simple. For instance, if a user wants to view the specialization hierarchy as an outline in MS-Word, she can write simple macros in MS-Word that use the object API to extract information from the Handbook and display it as an outline.

6.9 A Small and Transparent API

The object API is both small and transparent. We have made sure that there is minimal overlap in the functionality provided by the properties and methods that make up the API. This can be seen in the case of attributes. The reason why we did not implement 'Name' as a property is that Name is an attribute and it should be accessed exactly like all other attributes (system-generated and user-defined).

The API is also transparent. The server exposes OLE objects which could have been created in any of the popular programming languages. Therefore, the language used to develop the server is abstracted from the users of the API. Moreover, as mentioned

above, the API also hides the database schema. Thus both the code and the database schema are completely transparent to the user.

6.10 Support for full Dependency and Attribute Inheritance

The new implementation would provide full dependency and attribute inheritance (as explained in section 2.2.6). The inheritance algorithms in the object server are being coded from scratch. This avoids the legacy issues in reusing code written for the current implementation and allows us to implement all the features that were missing from the old algorithms.

6.11 Full Support for the Import and Export Processes

To ensure that processes can be exchanged between the Process Handbook and other process representation tools, we need to make sure that a client can use the object API to represent all the object in the PIF class hierarchy. Figure 25 shows the PIF class hierarchy. All of these objects can be represented in the new implementation of the Process Handbook by using entities and their attributes.

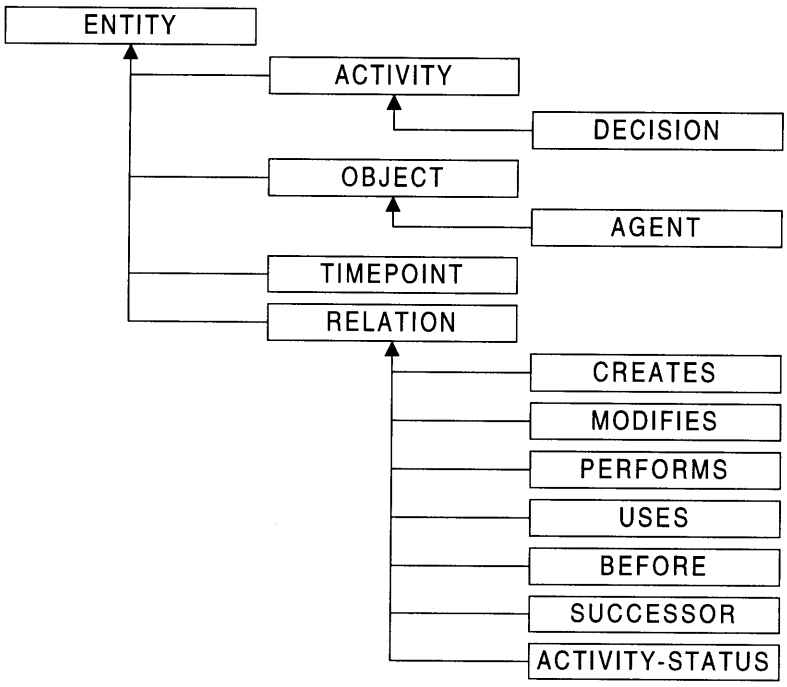


Figure 25. The PIF class hierarchy.

Both PIF Entity and Activity can be represented as the Process Handbook Entity object. The PIF Object and PIF Agent can be represented as resources in the Process Handbook. The PIF Decision can also be implemented as an activity with ports (see section 5.2.3). A PIF Timepoint can be implemented as a thing (entity type 0) in the Process Handbook with attributes that specify its nature. All of the PIF relations can also be implemented as attributes of an entity. For example, the PIF Creates can be implemented as an attribute on an activity that contains the ID of the resource created by the activity.

Chapter 7

Future Work

The Process Handbook server achieves many goals and provides all of the Process Handbook functionality. However, there are further additions that have to be made to make the server more efficient and useful.

First, the current server does not implement any kind of caching. This does not affect the performance of the initial implementation of the server because both the server and the database exist on the same machine. Therefore, all the interactions between the server and the database are over the local bus (which has very high-bandwidth and very low-latency) of the machine. However, as the size of the database grows, it might have to be shifted to a dedicated data server such as a Windows NT machine with the SQL Server. In case that happens, all the communications between the server and the database would be over a network link (which is considerably slower than the local bus of a machine). To make this data interchange efficient, it would be necessary to implement a cache at the server. This thesis did not deal with the implementation of the cache because implementing such a cache is extremely complicated because of the inheritance in the Process Handbook model.

Second, the security features designed by Calvin Yuen (Yuen, 1997) have to be merged with the object API. This will not change the basic interface provided by the object API, but would require extensions to the code in the server. The server would have to authenticate each transaction. This thesis did not deal with this merging process because both the object API and the access control mechanism API were being developed in parallel. Now that both these API's are fully specified, the object server has to be changed to use the access control mechanism and the object API has to be changed to reflect these changes.

Third, the object server needs to have algorithms that can perform automatic garbage collection in the database. Currently, the entities, relations, and attributes that are deleted are marked as deleted in the database, but they are not removed from the database. The server should have routines that can check the database at the end of each session and perform garbage collection. This would keep the database relatively small and all the queries to the database relatively fast.

Chapter 8

Conclusions

A three-tier client/server architecture has been used to design the Process Handbook. The clients used for the viewing and editing of the processes in the Process Handbook form tier 1, the inheritance and database interaction routines form tier 2 (also called the 'middleware'), and the underlying database forms tier 3. Tier 2 exposes OLE objects that can be manipulated by the tier 1 clients through an object API.

This object API abstracts both the Process Handbook algorithms and the schema of the underlying database. The abstraction of the complicated algorithms, especially the algorithms dealing with inheritance, allows developers to extend the existing functionality (by treating the available routines as abstract black-boxes) and create new clients (such as new GUI viewers and editors) very easily. The abstraction of the underlying database allows the Process Handbook developers to replace the existing database or change the database schema with minimal impact on the tier 1 clients.

The initial viewers that have been developed for testing the server (that is still being developed) prove that the new implementation will make the Process Handbook more extensible and robust. Any user can develop her own clients easily by using the object API. Thus a user can interact with the Process Handbook in whatever manner she finds most useful. Moreover, the users cannot corrupt the database through an inadvertent action. The API only lets the users make changes to the database which are allowed in the Process Handbook methodology.

Finally, implementing a new version of the Process Handbook has allowed us to augment the functionality provided by the previous versions. The new implementation provides full support for the inheritance of attributes and dependencies. The dependencies, which are the most important entities in the Process Handbook, are now represented exactly as the semantic model of the Process Handbook specifies. Thus, the new three-tier

client/server design will make the Process Handbook an extremely useful, extensible, and robust tool for analyzing processes.

References

Ahmed, Erfanuddin. (1995). *A Data Abstraction with Inheritance in the Process Handbook.* Unpublished M.S. thesis, Department of Electrical Engineering and Computer Science, MIT.

Berg, K. (1997). Component-Based Development: No Silver Bullet, *Object Magazine*, March 1997.

Chan, Frank Y. (1995). *The Round Trip Problem: A Solution for the Process Handbook.* Unpublished M.S. thesis, Department of Electrical Engineering and Computer Science, MIT.

Dellarocas, C. (1996). *A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components.* Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, MIT.

Elley, Yassir (1996). A Flexible Process Editor for the Process Handbook. Unpublished M.S. thesis, Department of Electrical Engineering and Computer Science, MIT.

Lee, J., Grunninger, M., Jin, M., Malone, T.W., Tate, A., Yost, G. and other members of the PIF Working Group (1996), The PIF Process Interchange Format and Framework Version 1.1, *Proceedings of the Workshop on Ontological Engineering*, ECAI 1996. Budapest, Hungary.

Malone, T.W. (1996). *How Will You Manage in the 21st Century? From Command and Control to Cultivate and Coordinate* (Discussion Paper), Center for Coordination Science, MIT.

Malone, T.W. and Crowston, K. (1994). The interdisciplinary study of coordination, *ACM Computing Surveys.*

Malone, T.W., Crowston, K., Lee, J., Pentland, B., Dellarocas, C., Wyner, G., Quimby, J., Osborne, C. and Bernstein, A. (1997). *Tools for inventing organizations: Towards a handbook of organizational processes* (Working Paper 198), Center for Coordination Science, MIT.

Nerson, Jean-Marc and Meyer, Bertrand. (1993). *Object-oriented applications.* Prentice Hall.

Spencer, K.L. and Miller, K. (1996). *Client/Server Programming with Microsoft Visual Basic.* Microsoft Press, Redmond, Washington.

Tanenbaum, A. (1992). *Modern Operating Systems.* Prentice Hall.

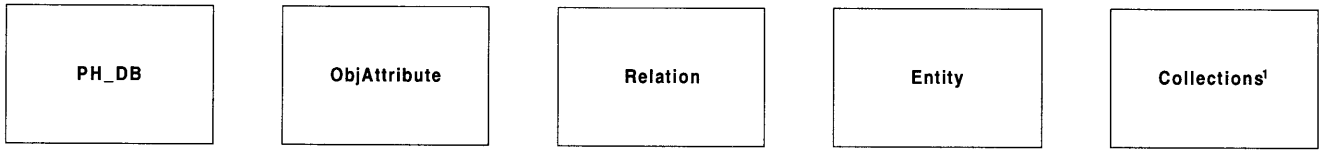
Yuen, Calvin M. (1997). *A Discretionary Access Control Policy for the Process Handbook.* Unpublished M.S. thesis, Department of Electrical Engineering and Computer Science, MIT.

__. (1995). *Microsoft Visual Basic: Building Client/Server Applications with Visual Basic.* Microsoft Corporation.

__. (1995). *Micorsoft Visual Basic: Professional Features.* Microsoft Corporation.

__. (1996). *Component Object Model Specification.* Microsoft Corporation. Available at http://www.microsoft.com/intdev/sdk/docs/com/comintro.htm.

APPENDIX A (The Object API Documentation)



Methods:
OpenDB
GetRootEntity
GetEntity
GetObjAttribute
GetRelation
GetEntityByPIFID
GetAttriOwnerEntities
CloseDB

Properties:
Name
DataType
Value
Object
Inherited
ID
Deleted
OwnerObjectID
InheritedFromAttrID
InheritanceBehavior
DefaultValue

Methods:
SetValueInContext
SetObjectInContext
DeleteInContext
GetAttributeType
SetAttributeType

Properties:
Name
RelationType
StartID
EndID
Inherited
ID
Deleted
ContextID
InheritedFromRelID

Methods:
GetStartPath
GetEndPath
DeleteInContext

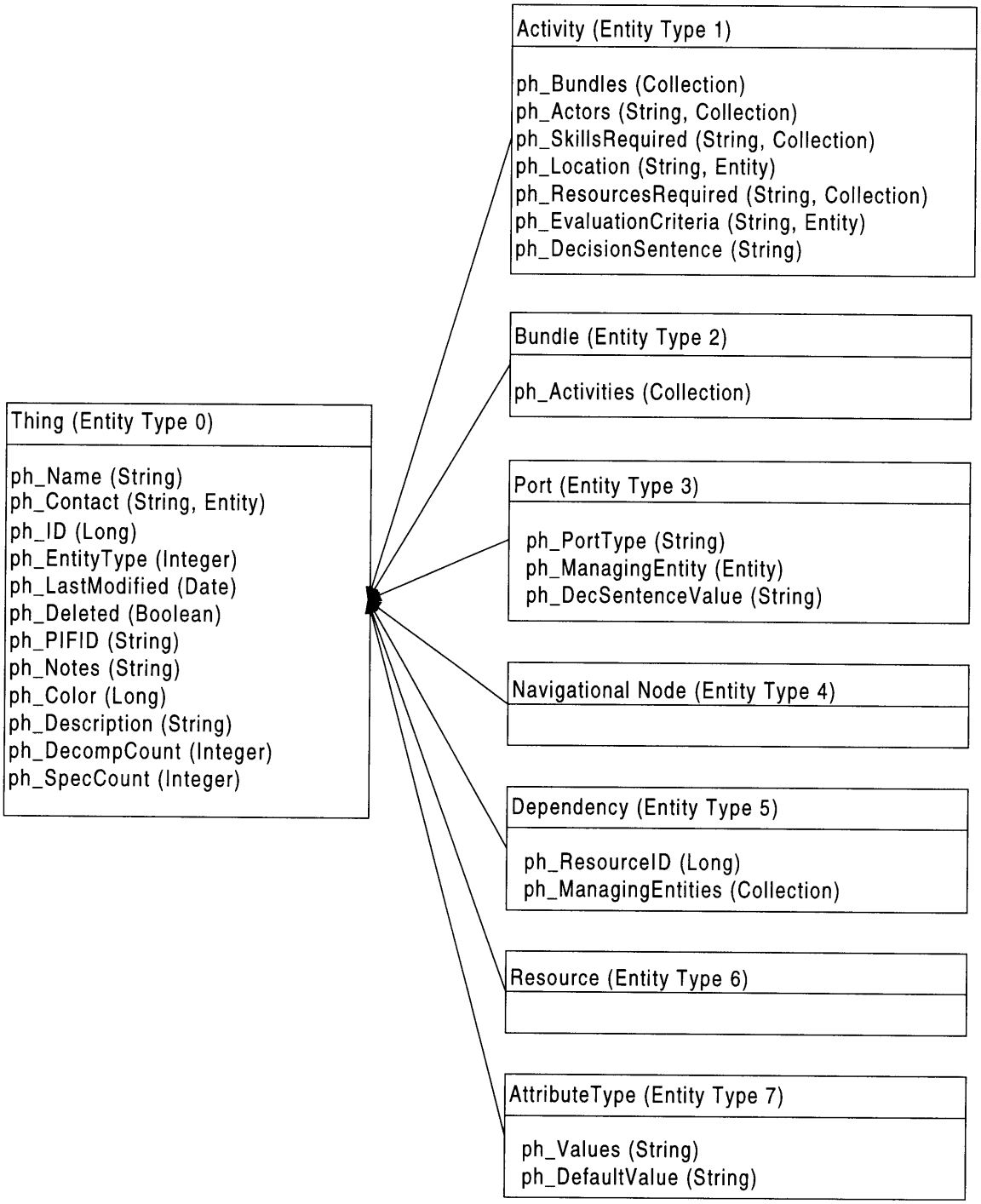
Methods:
GetAttribute
GetAttributes
AddNewAttribute
GetNavRelations
GetNavigatedFrom
AddNewNavRelation
GetSpecializations
GetGeneralizations
MoveToParent
RemoveFromParent
AdoptSpecialization
AddNewSpecialization
GetDecomposition
GetWhereUsed
AddDecomp
CreateConnector
GetConnectors
RemoveFromDecomp*
GetAttributeValue*
GetAttributeObject*
AddItemtoAttrCol*
RemoveItemFromAttrCol*
SetAttributeValue*
SetAttributeObject*
DeleteAttribute*
DeleteNavRelation*
ReplaceInDecomp*
DeleteConnector*

Methods
Count
GetItemByIndex
RemoveItemByIndex
AddItem

[1] These properties and methods apply to the three collection objects: ObjAttributes, Relations, and Entities.
* These methods are for efficiency in interaction with the server.

Properties and Methods for the First-Class Objects

All the System Generated Attributes of Thing (Entity Type 0)
are Inherited by all Other Types



The system-generated attributes for the Entity object.

THE FOLLOWING IS A LIST OF ALL THE METHODS AND PROPERTIES FOR THE FIRST CLASS OLE OBJECTS (AND COLLECTIONS) EXPOSED BY THE OLE SERVER

(Each method or property is first followed by an example of code that uses that method and a brief description. The 'Dim' keyword is used in Visual Basic to declare local variables. The errors that will be returned have not been finalized yet. They will be explained and documented after the server can perform the basic functions seamlessly.)

PH_DB

Properties:

None.

Methods:

OpenDB:

Dim myPH_DB as New PH_DB
.....
myPH_DB.OpenDB (FilePath, *UserID*, *Password*)

This method takes in a string that is the complete file path starting with the root directory (C:\ on typical Windows machines). The database file should be on the computer that has the ph_server. The second and the third argument are the userID and password required to use the server. The current implementation does not require these arguments. This method would return an informative error in case there is a problem with opening the database.

GetEntityByPIFID:

Dim myID as Long
Dim myType as Integer
.....
myID = myPH_DB.GetEntityByPIFID (PIFID)

This method takes in the unique ID of an object and returns the ID of that object in the current database. If the object does not exist in the current database, the method generates an error.

GetRootEntity:

Dim myEntity as New Entity

.....
Set myEntity = myPH_DB.GetRootEntity

This method returns the root entity from the Process Handbook database. The root entity that is returned by this method has entity type 0. The root entity for each of the entity types are the specializations of this entity. The database has the following entity types:

0 Thing
1 Activity
2 Bundle
3 Port
4 Navigational Node
5 Dependency
6 Resource
7 AttributeType

GetAttriOwnerEntities:

Dim myEntities as New Entities
.....
Set myEntities = myPH_DB.GetAttriOwnerEntities (AttributeName, *Optional* AttributeValue)

This method returns all the entities that have the attribute named AttributeName. The second argument to this method is optional. If this argument is specified then the method would return all the entities that have the attribute named AttributeName whose value is AttributeValue (this arument is a Variant).

For the attribute 'Name', the user can pass in any part of the name and the returned collection would contain all the entities whose name contains the string passed in as argument.

The user should keep in mind that some attributes (like 'Name' and 'Contact') belong to all entities. If the user tries to get all the entities that have a system generated attribute, the collection generated by the server can be huge.

GetEntity:

Dim myEntity as Entity
.....
Set myEntity = myPH_DB.GetEntity (EntityID)

This method returns the Entity whose ID is EntityID.

GetObjAttribute:

Dim myObjAttribute as ObjAttribute
.....
Set myObjAttribute = myPH_DB.GetObjAttribute (AttriID)

This method returns the ObjAttribute whose ID is AttriID.

GetRelation:

Dim myRelation as Relation
.....
Set myRelation = myPH_DB.GetRelation (RelID)

This method returns the Relation whose ID is RelID.

CloseDB:

Dim myPH_DB as New PH_DB
.....
myPH_DB.CloseDB

This method closes the database being currently used by the server.

ENTITY

Attributes (having non-object values) Generated by the System:

Attribute Name	Attribute Data Type	Attribute Value Status
ph_Name	String	Can Be Read and Updated
ph_Contact	String, Entity	Can Be Read and Updated
ph_ID	long	Can Only Be Read
ph_EntityType[1]	Integer	Can Only Be Read
ph_LastModified	Date	Can Only Be Read
ph_Deleted	Boolean	Can Be Read and Updated
ph_PIFID	String	Can Only Be Read
ph_Notes	String	Can Be Read and Updated
ph_Color	Long	Can Be Read and Updated
ph_Description	String	Can Be Read and Updated
ph_DecompCount[2]	Integer	Can Only Be Read
ph_SpecCount[3]	Integer	Can Only Be Read
ph_Actors[4]	String, Col. of Entities	Can Be Read and Updated
ph_SkillsRequired[4]	String, Col. of Entities	Can Be Read and Updated
ph_Location[4]	String, Entity	Can Be Read and Updated

ph_ResourcesRequired[4]	String, Col. of Resources (Entity Type 6)	Can Be Read and Updated
ph_EvaluationCriteria[4]	String, Entity	Can Be Read and Updated
ph_DecisionSentence[4]	String	Can Be Read and Updated
ph_Bundles[4]	Collection of bundles (EntityType 2)	Can Be Read and Updated
ph_Activities[5]	Collection of activities (EntityType 1)	Can Be Read and Updated
ph_ResourceID[6]	Long	Can Be Read and Updated
ph_ManagingEntities[6]	Collection of activities (Entity Type 1)	Can Be Read and Updated
ph_ManagingEntity[7]	Collection of ports (Entity Type 3)	Can Be Read and Updated
ph_PortType[7]	Integer	Can Be Read and Updated
ph_DecSentenceValue[7]	String	Can Be Read and Updated
ph_Values[8]	String	Can Be Read and Updated
ph_DefaultValue[9]	String	Can Be Read and Updated

[1] The following are the possible entity types:

0 Thing
1 Activity
2 Bundle
3 Port
4 Navigational Node
5 Dependency
6 Resource
7 AttributeType

[2] The number of other entities in this Entity's decomposition.

[3] The number of other entities in this Entity's specializations (this includes bundles).

[4] Only activities can have these attributes. The attribute ph_DecisionSentence holds a string expression that can be evaluated to a string, such as "true" or "false." This string is then used to pick between multiple alternatives.

[5] Only bundles have this attribute.

[6] Only the Dependencies (EntityType 5) can have ResourceID attribute (if the dependency does not have a resource then this attribute has the value -1) and ManagingEntityID attribute.

[7] These attributes only applies to entity type 3 (port). The attribute ph_PortType can have the following values:

"untyped"	No Port Type Specified (this is the type of the root port)
"consumer"	Consumer Port
"producer"	Producer Type

The attribute ph_DecSentenceValue contains a string expression which is used in conjunction with the ph_DecisionSentence attribute of decision activities to pick between multiple alternatives.

[8] This attribute only applies to entity type 7 (AttributeType) which is an enumeration of values. The value of this attribute is a comma delimited list of all the members of the enumerated AttributeType (eg. High, Medium, Low).

[9] This attribute only applies to entity type 3 (port). The value of this attribute is the default value with the AttributeType set.

Methods:

GetAttributes:

Dim myObjAttributes as New ObjAttributes
.....
Set myObjAttributes = myEntity.GetAttributes (*Optional* Deleted)

This method takes in an optional boolean argument. This Method just returns a collection of all the attributes of the Entity. If the argument is True then the deleted attributes are also added to the collection. If the argument is not provided, the deleted attributes are not returned. The attributes can then be taken out one by one by using the methods available for the Attributes object.

This method applies to all entity types.

GetAttribute:

Dim myObjAttribute as New ObjAttribute
.....
Set myObjAttribute = myEntity.GetAttribute (AttriName)

This method takes as an argument the string that is the name of the attribute. It returns that Attribute if the Entity has an attribute by this name. If no such attribute is found, the server raises an error.

This method applies to all entity types.

GetAttributeValue:

Dim myValue as Variant
.....
MyValue = myEntity.GetAttributeValue (AttriName)

This method gets the value of an attribute in one API call. The value of an attribute can also be determined in two API calls, by first getting the attribute object and then finding out its value. This method is primarily for efficiency (particularly for use with the system generated attributes). If no such attribute is found, the server raises an error.

This method applies to all entity types.

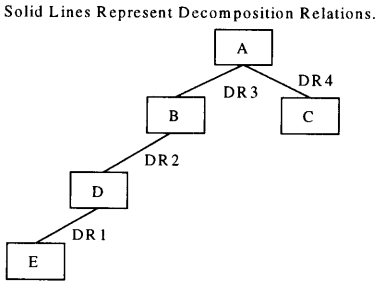
GetAttributeObject:

Dim myObject as New Object
.....
Set myObject = myEntity.GetAttributeObject (AttriName)

This method gets the object stored as an attribute in one API call. The object can also be determined in two API calls, by first getting the attribute object and then finding the object contained in it. This method is primarily for efficiency (particularly for use with the system generated attributes). If no such attribute is found, the server raises an error.

This method applies to all entity types.

Figure 1 (below) makes it easier to understand some of the following methods.



SetAttributeValue:

Dim Changed as Boolean

myEntity.SetAttributeValue (AttriName, NewValue, *Optional* ContextArray, *Optional* Changed, *Optional* ChangedArray)

This method is also primarily for efficiency. The method takes in a variant that is then stored as the value of the attribute named AttriName. If no such attribute is found, the

system raises an error. The last three arguments are optional and are used if the entity exists in a decomposition context. If the Entity is inherited, then we need to create a new Entity. The third argument is an array of longs that the client passes to the server. This array contains the ID's of all the decomprelations starting from the current Entity and ending at the context Entity. For example, if we were trying to add an attribute to E (see Figure 1), the array would contain the ID's of DR1, DR2, and DR3 (in that order). The last two arguments are passed in by reference. Other than returning the new attribute that is created, this method changes the value of the last two arguments. The argument 'Changed' is set to true if a new Entity has been created due to the addition of the new attribute. This creation of a new Entity might also result in the creation of a number of other new Entities (for example, if the decomposition relation between A and B was inherited, this change would also create new Entities that would replace B and D). The last argument is an array of tuples (a 2 dimensional array) that contains tuples of the old and new ID's. For example, if the above case was true, the first tuple would consist of the ID of B and the ID of the new Entity that replaces B in this decomposition, the second tuple would deal with D and its replacement, and the third tuple would contain the ID's of E and its replacement. For any changes in context, the user has to provide all three optional arguments otherwise the server would generate an error.

This method is applicable to all entity types.

NOTE: THE ARGUMENTS *ContextArray*, *Changed*, AND *ChangedArray* ARE USED IN ALL THE METHODS THAT CAN BE MADE IN CONTEXT. FOR THE SAKE OF CONCISENESS, THESE ARGUMENTS WILL NOT BE EXPLAINED IN DETAIL AGAIN.

SetAttributeObject:

Dim Changed as Boolean
.....
myEntity.SetAttributeObject (AttriName, NewObject, *Optional* ContextArray, *Optional* Changed, *Optional* ChangedArray)

This method is also primarily for efficiency. It is very similar to the SetAttributeValue method. The only difference is that in this case the client provides an object that has to be stored in the attribute.

This method is applicable to all entity types.

AddItemtoAttrCol:

.....
myEntity.AddItemtoAttrCol (AttriName, Object)

This method is applied to attributes that have a collection stored in them (for example 'bundles' attribute for activities and 'activities' attribute for bundles). The object that is passed in as the second argument is added to the collection. For instance if the user wants to add a new bundle to the bundles under that activity, the user just creates the bundle and then passes it in as the second argument to this method. The first argument is the name of the attribute.

RemoveItemfromAttrCol:

.....
myEntity.RemoveItemfromAttrCol (AttriName, Index)

This method is the converse of the previous method. This method removes the object (whose index into the collection is passed in as the second argument to this method) from the collection stored in the attribute.

AddNewAttribute:

Dim myObjAttribute as New ObjAttribute
.....
Set myObjAttribute = myEntity.AddNewAttribute(AttriName, *Optional* AttriValue, *Optioanl* AttriObject, *Optional* ContextArray, *Optional* Changed, *Optional* ChangedArray)

This Method takes in six arguments. The first is a string that is the name of the new attribute to be added to this Entity. The AttriName cannot start with "ph_" because only system generated attributes start with "ph_". The second argument is a variant that is the value of the attribute. If the attribute contains an object then the user just uses the third argument which is the object that this attribute will contain. The user has to use either argument two or argument three and has to specify the argument that is being passed in. The last three arguments are used if the new attribute is being added in a decomposition context. The user also has to specify these three arguments by name (because the user would skip either argument two or three) and has to pass in all three argument if the changes are being made in a decomposition context. They are the same as the similarly named arguments in the SetAttributeValue method. This Method returns the newly created Attribute object.

This method is applicable for all entity types.

DeleteAttribute:

.....
myEntity.DeleteAttribute(AttriName, *Optional* ContextArray, *Optional* Changed, *Optional* ChangedArray)

This Method takes in four arguments. The first is a string that is the name of the attribute that is to be deleted. The last three arguments are used if the new attribute is being added in a decomposition context. They are the same as the similarly named arguments in the SetAttributeValue method.

This method is applicable for all entity types.

GetSpecializations:

Dim myEntities as New Entities
.....
Set myEntities = myEntity.GetSpecializations (*Optional* Deleted)

This method just returns a collection of Entities that are the specializations of this Entity. These Entities can then be taken out one by one using the methods for the Entities Object. If the Optional Boolean argument is provided and is true then the deleted specialization Entities are also added to the collection.

This method is applicable for all entity types. If this method is applied to the root, this method would return the roots of all the entity types. For all other entity types, the entity type of all the entities in the returned collection are the same as the entity type of the entity to which this method is applied.

GetGeneralizations:

Dim myEntities as New Entities
.....
Set myEntities = Entity.GetGeneralizations (*Optional* Deleted)

This method just returns a collection of Entities that are the generalizations of this Entity. These Entities can then be taken out one by one using the methods for the Entities Object. If the Optional Boolean argument is provided and is true then the deleted generalizations are also added to the collection.

This method is applicable for all entity types. The entity type of all the entities in the returned collection are the same as the entity type of the entity to which this method is applied.

MoveToParent:

.....
myEntity.MoveToParent (NewParentEntity, TestOnly)

This method takes in another Entity object and moves the current Entity to be a specialization of this new Entity. The second argument is a boolean that tells the server

that the client only wishes to check if this move is valid. If the TestOnly argument is true then the method executes in the following manner: If the move is invalid, informative error is generated (the method uses the errors as a signaling mechanism). If the move is valid then the method executes without any error.

If the TestOnly argument is false then the method is executed in the following manner: The same errors are generated if the move is invalid. However, if the move is valid then the move operation is perfomed.

This method is applicable for all entity types. For the entities of type 0, we can move them to any of the other entities (and they would be changed to the parent entity type). For all other entity types, the entity type of the NewParentEntity has to be the same as the entity type of myEntity.

RemoveFromParent:

```
Dim Del as Boolean
.....
Del = myEntity.RemoveFromSpec (ParentEntity)
```

This method removes the Entity from the specialization of ParentEntity. If ParentEntity is the only parent of the Entity then the Entity is deleted and the method returns true. If the Entity is not deleted then the method returns false.

This method is applicable for all entity types.

AddNewSpecialization:

```
Dim myNewEnt as New Entity
.....
Set myNewEnt = myEntity.AddNewSpec(NameStr, ContactStr)
```

This method takes in two strings. The name of the new Entity that has to be created, and the contact name for this Entity. This method requires that the name is a non-empty string. The method returns the newly created Entity object.

This method is applicable for all entity types. The entity type of the result Entity (myNewEnt) is the same as the entity type of myEntity.

AdoptSpecialization:

```
.....
myEntity.AdoptSpecialization (otherEntity)
```

This method takes in the entity that has to be adopted. This method adds that entity to the specializations of myEntity (the entity that is adopted is not removed from its previous parents). If the object cannot be adopted then this method generates an informative error.

This method is applicable for all entity types. The entity type of otherEntity should be the same as the entity type of myEntity.

GetDecomposition:

Dim myRels as New Relations
.....
Set myRels = myEntity.GetDecomposition (*Optional* Deleted)

This method returns a collection of relations that make up the decomposition of myEntity. The returned relations can have different types (as explained in the table below). The client would then have to check for the different kinds of Relations to separate them and display them. If the Optional Boolean argument is provided and is true then the deleted relations in the decomposition are also added to the collection.

Entity type of myEntity	Relation types for the returned relations (myRels). See documentation for the Relation object.
0	relation type 1 (between entity types 0 and 0)
1	relation type 1 (between entity types 1 and 1, between entity types 1 and 3, and between entity types 1 and 5) and relation type 3 (between entity types 3 and 3)
2	relation type 1 (between entity types 2 and 2)
3	relation type 1 (between entity types 3 and 3)
4	relation type 1 (between entity types 4 and 4)
5	relation type 1 (between entity types 5 and 5 and between entity types 5 and 3)
6	relation type 1 (between entity types 6 and 6)
7	relation type 1 (between entity types 7 and 7)

GetWhereUsed

Dim myDecRels as New Relations
.....
Set myDecRels = myEntity.GetWhereUsed(*Optional* Deleted)

This method returns a collection of relations (that have this Entity as the sub-Entity). If the Optional Boolean argument is provided and is true then the deleted decomposition Entities are also added to the collection.

Entity type of myEntity	Relation types for the returned relations (myRels). See documentation for the Relation object.
0	relation type 1 (between entity types 0 and 0)
1	relation type 1 (between entity types 1 and 1)
2	relation type 1 (between entity types 2 and 2)
3	relation type 1 (between entity types 1 and 3, between entity types 5 and 3, and between entity types 3 and 3)
4	relation type 1 (between entity types 4 and 4)
5	relation type 1 (between entity types 5 and 5 and between entity types 1 and 5)
6	relation type 1 (between entity types 6 and 6)
7	relation type 1 (between entity types 7 and 7)

RemoveFromDecomp:

Dim Changed as Boolean
.....
Changed = myEntity.RemoveFromDecomp(ParentEntity, ContextArray, ChangedArray)

The Entity here is the Entity that the client wants to remove from the decomposition. The ParentEntity is the immediate parent of the Entity in the decomposition from which the client wants to remove it. Changed, ContextArray, and ChangedArray are the same as the similarly named arguments in the SetAttributeValue method. This method will generate an error if the ParentEntity is not the parent Entity in the database (removing the Entity from decomposition does not delete the Entity from the database).

Entity type of myEntity	Entity type of ParentEntity
0	0
1	1
2	2
3	1, 3, 5
4	4
5	1, 5
6	6
7	6

ReplaceInDecomp:

Dim Changed as Boolean
.....
Changed = myEntity.ReplaceActiInDecomp(ChildEntity, NewChildEntity, ContextArray, ChangedArray)

This method takes in four arguments. The first is the child that we want to replace in the decomposition. The second is the new entity that we want to put in the decomposition in place of the Child entity. Changed, ContextArray, and ChangedArray are the same as the similarly named arguments in the SetAttributeValue method. If the NewEntity cannot be added to the decomposition, then a helpful error is generated.

Entity type of myEntity	Entity type of ChildEntity	Entity type of NewChildEntity
0	0	0
1	1, 3, 5	1, 3, 5
2	2	2
3	3	3
4	4	4
5	5, 3	5, 3
6	6	6
7	7	7

AddDecomp:

Dim Changed as Boolean
Dim NewDecRelation as Relation
.....
Set NewDecRelation = myEntity.AddDecomp(NewEntity, ContextArray, ChangedArray, Changed)

This method takes in four arguments. The first is the entity that has to be added to the decomposition of this Entity. The other three arguments are the same as the similarly named arguments for the SetAttributeValue method. This method returns the new decomposition relation that is created due to the addition of the new decomposition child.

Entity type of myEntity	Entity type of NewEntity
0	0
1	1, 3, 5
2	2
3	3
4	4
5	5, 3
6	6
7	7

CreateConnector:

Dim myRel as New Relation

Dim Changed as Boolean
.....
Set myRel = myEntity.CreateConnector(EntityStartPort, EntityEndPort, Changed, ChangedArray, ContextArray)

This method creates a connector in the context of myEntity. The method takes the starting port entity (entity type 3) and the ending port entity (entity type 3) for this connector. The newly created connector is returned by the method. The last three arguments perform the same function as the similarly named arguments for the SetAttributeValue method. The creation of a connector might result in a number of other changes.

This method only applies to entity type 1.

DeleteConnector:

Dim Changed as Boolean
.....
Changed = myEntity.DeleteConnector(Connector, ContextArray, ChangedArray)

The Entity here is the Entity that is one of the end points for the connector. The Entity can either be the port that the connector is actually connected to or the activity to which that port belongs (thus, this method can only apply for entity types 1 and 3). The first argument is the connector object that the user wants to remove. Changed, ContextArray, and ChangedArray are the same as the similarly named arguments in the SetAttributeValue method. This method will generate an error if the connector that is being removed is not connected to the entity.

GetConnectors:

Dim myCons as New Relations
.....
Set myCons = myEntity.GetConnectors (ContextEntity)

This method is returns all the connectors that connect this myEntity with other entities in the context of ContextEntity. In case myEntity is an Activity (entity type 1) or a dependency (entity type 5), the connectors that are returned are the connectors that attach the ports of myEntity with other ports.

This method is only applicable to entity types 1, 3, and 5.

GetNavRelations:

Dim NewRelations as New Relations

.....
Set NewRelations = myEntity.GetNavigationalRelations

This method returns all the navigational relations that start at this Entity.

This method is applicable for all entity types.

GetNavigatedFrom:

Dim NewRelations as New Relations
.....
Set NewRelations = myEntity.GetNavigatedFrom

This method returns all the navigational relations that end at this Entity.

This method is applicable for all entity types.

AddNewNavRelation:

.....
myEntity.AddNewNavRelation(NewNavChildEntity)

This method adds a new navigational link from this Entity to the object whose ID is passed in as an argument.

This method is applicable for all entity types. The NewChildEntity can also have any entity type.

DeleteNavRelation:

.....
myEntity.DeleteNavRelation(NavChildEntity)

This method removes the navigational realtion between myEntity and NavChildEntity. If such a navigational relation does not exist then the server raises an error.

This method is applicable for all entity types. The NewChildEntity can also have any entity type.

OBJATTRIBUTE

Properties:

Property	Value Data Type	Status
Name	String	Can Be Read and Updated

DataType	String	Can Only Be Read
Value	Variant	Can Be Read and Updated
Inherited	Boolean	Can Only Be Read
ID	Long	Can Only Be Read
Deleted	Boolean	Can Only Be Read
InheritedFromAttrID	Long	Can Only Be Read[1]
OwnerObjectID	Long	Can Only Be Read[2]
InheritanceBehaviour	Integer	Can Be Read and Updated[3]
DefaultValue	Variant	Can Be Read and Updated

[1] This property will be -1 if the inherited property is false (the attribute is not inherited).

[2] This ID can be for a Resource, NavNode, Entity, Port, Bundle, or Folder. To find out the correct type, use the GetType method for the db object.

[3] This property can have the following values

1 No Inheritance (the specializations of the owner entity do not inherit this attribute)
2 Full Inheritance (the specializations of the owner entity inherit both the attribute and its value)
3 Default-value Inheritance (the specializations of the owner entity inherit this attribute but the value of the inherited attribute is the default value for this attribute)
4 Only Slot Inheritance (the specializations of the owner entity inherit the attribute but they only inherit the slot; the value of the inherited attribute is null)

Methods:

Bind:

```
Dim myObjAttribute as New ObjAttribute
.......
myObjAttribute.Bind (attriID)
```

This method binds the myObjAttribute object to the attribute whose ID is attriID.

SetValueInContext:

```
Dim Changed as Boolean
........
Changed = myObjAttribute.SetValueInContext (NewValue, ContextArray, ChangedArray)
```

This method renames the Entity in a context. Changed, ContextArray, and ChangedArray are similar to those in AddNewAttrInContext method for the Entity object. The

ContextArray starts with the ID of the decomposition parent of the Entity that has this attribute. In case this attribute belongs to a dependency, the ContextArray starts with the ID of the Closest Common Ancestor of the dependency.

DeleteInContext:

Dim Changed as Boolean
.....
Changed = myObjAttribute.DeleteInContext (ContextArray, ChangedArray)

This method deletes the attribute in a particular context. Changed, ContextArray, and ChangedArray are the same as those in the NewValueInContextMethod.

GetAttributeType:

Dim myAttrType as New AttributeType
.....
Set myAttrType = myObjAttribute.GetAttributeType

This method returns the AttributeType object that defines the set of values that this particular attribute can take. If there is no AttributeType for this particular attribute, the server generates an informative error.

SetAttributeType:

Dim Changed as Boolean
.....
Changed = myObjAttribute.SetAttributeType (myAttributeType, ContextArray, ChangedArray)

This method sets myAttributeType as the AttributeType of this particular ObjAttribute. This method changes the attribute so we need to pass in the context for the change. Changed, ContextArray, and ChangedArray are the same as those in the NewValueInContext method.

RELATION

Properties:

Property	Value Data Type	Status
Name	String	Can Be Read and Updated[1]
RelationType[2]	Integer	Can Only Be Read
StartID	Long	Can Only Be Read
EndID	Long	Can Only Be Read
Inherited	Boolean	Can Only Be Read

ID	Long	Can Only Be Read
Deleted	Boolean	Can Only Be Read
ContextID[3]	Long	Can Only Be Read
InheritedFromRelID	Long	Can Only Be Read

[1] This property has special meaning for decomposition relations. For instance if an entity exists in the decomposition of another entity more than once, the decomposition relation name allows the user to differentiate between the two.

[2] The type can be one of the following:
1 Decomposition Relation
2 Navigational Relation
3 Connector

[3] This property will hold the ID of the Entity in whose context this relation exists. For decomposition relations, the ContextID will be the same as the ParentID.

Methods:

GetStartPath:

```
Dim Path() as Long
........
myRelation.GetStartPath Path()
```

This method returns the ID's of all the decomposition relations starting from the port that is the starting point for a connector and ending at the context entity. The array that is passed in as the argument is updated and resized by the server.

GetEndPath:

```
Dim Path() as Long
........
myRelation.GetEndPath Path()
```

This method returns the ID's of all the decomposition relations starting from the port that is the ending point for a connector and ending at the context entity. The array that is passed in as the argument is updated and resized by the server.

DeleteInContext:

```
Dim Changed as Boolean
........
Changed = myRelation.DeleteInContext (ContextArray, ChangedArray)
```

This method deletes the relation in a particular context. This method is only valid for connectors and decomposition relations because the navigational nodes do not exist in context. Changed, ContextArray, and ChangedArray have the same meaning as the similarly named arguments for the NewValueInContext method for the attribute object. The ContextArray starts with the decomposition ID of the current decomposition relation (in case the relation is a decomposition relation) or the decomposition relations starting from the context activity of the connector (in case the relation is a connector).

THE FOLLOWING ARE THE METHODS AND PROPERTIES FOR ENTITIES, OBJATTRIBUTES, AND RELATIONS.
(THE REASON FOR THE METHODS AND PROPERTIES BEING THE SAME IS THAT ALL THESE OBJECTS ARE IMPLEMENTED USING THE COLLECTION OBJECT PROVIDED BY VISUAL BASIC).
IN THE METHODS BELOW COLLECTION CAN BE REPLACED BY ANY OF THE ABOVE OBJECTS.

Properties:

None.

Methods:

Count:

```
Dim myCount as Integer
.....
myCount = Collection.Count
```

This method just returns the number of objects that are in the collection of objects. The Count starts from 1.

AddItem:

```
.....
Collection.AddItem (ObjectID)
```

The method takes in an object's ID (for example, the ID of an Entity that the client wants to add to an existing Entities object) and adds it to the collection.

GetItemByIndex:

```
Dim myObject as Object        ' here object can be Entity, Resource, NavNode etc.
.....
Set myObject = Collection.GetItemInd (Index)
```

This method takes in the integer argument Index and returns the object that has that index in the collection. The indices start from 1. The collection is unchanged by this method.

RemoveItemByIndex:

```
Dim myObject as Object        ' here object can be Entity, Resource, NavNode etc.
.....
Set myObject = Collection.RemoveItemInd (Index)
```

This method takes in the integer argument Index and returns the object that has that index in the collection. The indices start from 1. The object that is returned is removed from the collection.